José Luís Vaz Ribeiro

Orquestração e Composição de Serviços Web Usando BPEL

José Luís Vaz Ribeiro

Orquestração e Composição de Serviços Web Usando BPEL

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Dr. Joaquim Arnaldo Martins, Professor Catedrático do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Dedico este trabalho a todos os meus amigos, por me incentivarem a trabalhar e pelas horas de diversão nos intervalos do trabalho, e aos meus pais e irmãos, por me terem aturado nestes anos de vida académica.

o júri

Presidente

Prof. Dr.ª Maria Beatriz Alves de Sousa Santos

Professora Associada com Agregação da Universidade de Aveiro

Vogais

Prof. Dr. Joaquim Arnaldo Martins

Professor Catedrático da Universidade de Aveiro

Prof. Dr. Fernando Joaquim Lopes Moreira

Professor Auxiliar do Departamento de Inovação, Ciência e Tecnologia da Universidade Portucalense

Agradecimentos

Quero agradecer em primeiro lugar ao meu orientador, o professor Joaquim Arnaldo Martins, pelo apoio dado durante a realização desta dissertação, desde a escolha do tema até aos detalhes finais da entrega, tendo-me sempre incentivado e dado novas perspectivas sempre que me senti mais perdido ou com dúvidas.

Quero agradecer também ao co-orientador, Marco Fernandes, por ter estado disponível quando precisei dele e por ter ajudado a descobrir alguns dos erros que surgiram durante a realização de processos práticos.

Agradeço aos meus pais, pelo apoio que sempre me deram durante a vida académica, quer moral quer monetário, pois sem eles não teria feito o curso; e por terem apoiado nos momentos difíceis que surgiram ao longo deste. Por tudo isso, e por terem acreditado em mim, o meu muito sincero obrigado.

Aos meus irmãos, Inês e Mário, por me terem aturado estes anos, em especial a Inês, pelo apoio que deu, pelas refeições que cozinhou, pelos incentivos que deu, e pela paciência que teve comigo, que nem sempre fui o mais fácil dos irmãos. Ao Mário, porque quando quer, é um irmão espectacular, e até tem querido bastantes vezes.

Ao Telmo e ao Marco, por terem disponibilizado a casa para estudo, para diversão e para tudo o resto que um estudante precisa. Marco, obrigado pela ajuda nos trabalhos e desculpa lá se não fui às vezes o melhor colega de grupo que podia ser, mas que tentei sempre. Telmo, por teres conseguido meter nestas cabeças um bocado de responsabilidade e por estares lá sempre que precisávamos, seja para explicar uma função num trabalho, ou um jogador para a equipa de “Mario Strikers”.

Ao Pedro Lobo, por me ter dado tanto tanto tanto na cabeça quando não apetecia trabalhar, pela ajuda que deu sempre que precisávamos, por ter sido um tolo na altura da diversão, e um senhor na altura de trabalhar, e por ser o amigo que é, mesmo quando também ele estava enterrado na sua dissertação.

Ao Pinelas, ao André, ao Bruno, ao Tiago, à Vânia, ao Pedro, ao Nelson, ao Jerónimo, ao Filipe, e a todos os meus amigos e colegas que não nomeio, mas

que sabem que estou a pensar neles, pelo suporte que deram, pela força, e por serem os melhores amigos que se pode desejar. Um abraço enorme para todos.

Ao Jorge, DJ, Gil, Acácio, Popas, Rodas, Miguel e toda a malta de “Magic”, pelos bons momentos de descontracção e diversão, que também foram importantes para descomprimir do trabalho.

E finalmente, um agradecimento muito especial para a Cláudia, por ter sido a pessoa que mais acreditou em mim, mais me deu na cabeça, mais me incentivou a trabalhar e mais me animou quando precisei de ânimo. Por te teres preocupado tanto comigo, e por seres uma querida no geral e uma tolita em particular, aqui fica um abraço especial.

Palavras-chave Serviço, Web, Orquestração, Arquitectura, Comunicação, Composição, *Servlet*, Ambiente Gráfico, Processo Negócio, Actividade, Domínio.

Resumo

Este trabalho pretende expor de uma forma simples e clara os conceitos associados às Arquitecturas Orientadas ao Serviço, dando uma visão da evolução deste paradigma de programação, tecnologias existentes, ferramentas e motores disponíveis para o desenvolvimento, uso e teste de *Web Services*.

O principal foco será a linguagem BPEL (*Business Process Execution Language*), que é considerada como o standard para a orquestração de *Web Services*, sendo o trabalho realizado tendo por base esta linguagem.

Keywords

Service, Web, Orchestration, Composition, Arquitecture, Servlet, Graphical Environment, Business Process, Activity, Scope.

Abstract

This work intends to expose, in a simple and clear way, the concepts associated with Service Oriented Architectures, providing a vision of the evolution of this information paradigm, the existing technologies, the applications, tools and engines available to develop, use and test of Web Services in programming business processes.

The main focus will be in BPEL (Business Process Execution Language), considerate to be the standard in programming languages for orchestration of Web Services, being the base to this work.

Índice

Índice de Ilustrações:

Índice de tabelas:

Capítulo 1 – Introdução

As tecnologias de informação sofreram uma grande expansão com o aparecimento e posterior massificação da Internet, ocupando actualmente um lugar de destaque na vanguarda da evolução tecnológica. As trocas de conhecimentos e serviços tomaram uma complexidade e grandeza cada vez maior, ao mesmo tempo que o acesso à informação se tornou muito mais fácil e rápido, um recurso cujo potencial a industria depressa se apercebeu e cada vez mais usa.

Neste contexto, surgiram os *Web Services* [1], uma nova forma de pensar nos serviços e processos de negócio, em que estes passaram a ser disponibilizados dinamicamente e de um modo distribuído, sendo consumidos do mesmo modo. As empresas passaram a dispor assim de meios para usar recursos distribuídos na realização das suas tarefas/funções, isto é, ao invés de apenas poderem contar com os meios próprios em termos de computação e recursos informáticos, passaram a utilizar recursos, programas ou funções cuja execução e suporte não dependem dessas empresas, mas das entidades que disponibilizam os serviços. Os *Web Services* aproveitaram tecnologias já existentes de descrição, transporte, gestão, etc, e levaram ao desenvolvimento de outras, sendo algumas delas dedicadas à orquestração, isto é, à gestão e conjunção de serviços de modo a criar processos de negócio complexos.

Neste trabalho, pretende-se mostrar as soluções de orquestração de *Web Services*, em particular usando a linguagem WS-BPEL (Web Services *Business Process Enterprise Language*), ou simplesmente BPEL [2], com uma descrição do ambiente de utilização, das ferramentas mais utilizadas e de exemplos de utilização.

Assim, no capítulo 2, começa-se por se fazer uma Introdução às Arquitecturas Orientadas ao Serviço (SOA), o paradigma em que esta tecnologia é baseada, com uma visão histórica deste, assim como as suas principais características e ideias. São também abordadas as matérias relacionadas com

Web Services, a sua história, as necessidades que levaram ao seu aparecimento, o seu papel na divulgação das SOA's e tecnologias em que se baseia.

O capítulo 3 descreve a linguagem BPEL, a sua sintaxe, a sua história e evolução, a sua utilidade no âmbito dos *Web Services*, assim como explica as diferenças entre uma orquestração e uma coreografia na gestão e implementação de um processo de negócio que utilize *Web Services*.

No capítulo 4 são descritas três dos principais motores e aplicações usados para implementar processos de negócio em BPEL (Apache ODE [3], Netbeans [4] e ActiveBpel [5]). O seu funcionamento, a sua estrutura e a maneira como são aplicados.

O capítulo 5 apresenta as várias actividades disponíveis para a construção de processos de negócio em BPEL, isto é, a sua sintaxe, aplicada à ferramenta ActiveBPEL. Nele são descritas as aplicações das actividades, a sua aparência gráfica e uma ideia geral do seu uso e utilidade.

No capítulo 6 são analisados vários processos de negócio escritos em BPEL. Estes processos demonstram a utilização de quase todos os tipos de actividades presentes na linguagem, sendo analisados não só do ponto de vista gráfico, como do código fonte que lhe serve de base.

No capítulo 7 são escritas as conclusões da dissertação, o trabalho futuro a desenvolver neste âmbito e uma análise do que representa o BPEL no mundo das arquitecturas orientadas ao serviço e dos *Web Services*.

Capítulo 2 – Arquitecturas Orientadas ao Serviço (SOA) e Web Services

2.1 – SOA

O conceito associado às SOA (*Service Oriented Architecture*), de um paradigma para melhorar a flexibilidade dos sistemas informáticos, apesar de não ser novo (as ideias básicas foram introduzidas com as tecnologias CORBA (*Common Object Request Broker Architecture*) [6], DCOM (*Distributed Component Object Model*) [7] e DCE (*Enterprise Services Bus*) [8], entre outras)[9], pode ser considerado como o modelo dos anos 2000, a evolução natural dos modelos de desenvolvimento/programação. À programação orientada a objectos dos anos 80 seguiu-se o modelo baseado em componentes dos anos 90 até chegar a este modelo orientado ao serviço, que retém muitos dos benefícios do modelo de componentes (auto-descrição, encapsulação, "*Dinamic Descovery and Loading*"), mas que se destaca deste por passar da invocação remota de métodos e objectos para a troca de mensagens entre serviços, o que promove interoperabilidade e adaptabilidade entre serviços, já que as mensagens são enviadas sem precisarem de se preocupar com a maneira como são recebidas e tratadas no destino.

Não se pode considerar, no entanto, uma SOA como um produto ou tecnologia, ou mesmo um fim a alcançar para um processo de negócios (*business process* - a interacção entre dois negócios ou entre dois elementos de um negócio [10]) ou aplicação em geral; trata-se de uma arquitectura, não de uma aplicação desta, isto é, descreve e fornece os meios para um fim, mas não é, como os WS, um meio em si para ser aplicado. Isto leva a que sejam muitas vezes confundidos as definições de SOA com WS, que, embora seja a forma mais comum de implementação de SOA, não é a única. Exemplos de outras tecnologias usadas

para implementar SOA são DCOM [11], SOAP (*Simple Object Access Protocol*) [12], REST (*Representational State Transfer*)[13], CORBA, RPC (*Remote Procedure Call*) [14], entre outras.

As SOA têm por base em três grandes conceitos técnicos: serviços, interoperabilidade através de um barramento (*bus*) de serviços empresariais e *loose coupling*.

Um serviço é uma funcionalidade de negócio, auto-contida, com uma complexidade variável. Pode ser simples, como armazenar a informação de um cliente; ou complexa, como um processo de negócio completo para tratar uma ordem/compra de um cliente, fazendo a ligação entre o negócio e a tecnologia (IT – tecnologia de informação) que o suporta.

O *bus* de serviços empresariais (ESB, *Enterprise Services Bus* [15]) é a infra-estrutura que permite a interoperabilidade entre sistemas distribuídos para serviços, facilitando a distribuição dos processos de negócios entre múltiplos sistemas usando diferentes tecnologias e plataformas.

Loose coupling é o conceito que se refere à redução de dependências de sistema. Este factor torna-se ainda mais importante num cenário de SOA, em que processos de negócios são distribuídos por múltiplos destinos, e em que é ainda mais necessário que eventuais erros e modificações tenham um impacto reduzido no funcionamento dos processos que os usam, sob pena de uma falha de sistema afectar todos os outros sistemas, com consequências desastrosas. O inconveniente associado ao *loose coupling* reside no facto de a complexidade de desenvolvimento, manutenção e *debug* de sistemas distribuídos com esta propriedade aumenta consideravelmente.

A nível empresarial, adopção de SOA numa companhia implica assim mudanças profundas, uma vez que a introdução de uma nova funcionalidade deixa de ser resolvida com a designação de um departamento para uma dada tarefa mas sim com um conjunto de múltiplas tarefas de vários serviços, com estes e cada equipa ligada a estes a colaborar para implementar a funcionalidade. É necessária para isso a definição clara dos vários papéis, políticas e processos, como a definição do ciclo de vida do serviço ou o modelo a usar no desenvolvimento de serviços, entre outros [16].

Com esta evolução, surgiu a necessidade de uma nova função dentro de uma empresa: o Arquitecto de Software. Este, para além das funções habituais num programador, também tem a capacidade de analisar e conhecer o modelo de negócio pretendido, adaptando a arquitectura para as suas especificações próprias. Esta evolução do técnico ou engenheiro informático surgiu assim das necessidades específicas levantadas pela adopção de uma SOA nas empresas.

Resumindo, SOA é, assim, um paradigma para organizar e utilizar capacidades distribuídas controladas por diferentes entidades, providenciando as directrizes para a criação de um meio uniforme de oferecer, descobrir, interagir e usar essas capacidades para obter os efeitos desejados [17].

2.2 – Web Services

"*A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards*"[18].

A implementação de uma SOA pode ser realizada através utilização distribuída de serviços via Web, designados por *Web Services* (WS) - por definição, um WS é assim uma aplicação/software auto-contida, distribuída em rede, normalmente a Internet, que providenciam uma funcionalidade a quem o acede, com uma interface pública e independente da plataforma em que é invocada, comunicando por mensagens de XML (*Extensible Markup Language*) [19] com o cliente [18], o que torna esta tecnologia como um dos meios ideais para implementar os conceitos das SOA.

Segundo este conceito, os programadores ("*software developers*") pesquisam e usam serviços da Internet (*Web Services*), de uma forma dinâmica, de modo a produzir os efeitos desejados mas sem despender tempo e recursos computacionais próprios para esse efeito, uma vez que esse esforço é realizado pelas companhias que os fornecem. Assim, o programador apenas tem que se preocupar em invocar um WS, fornecendo-lhe os dados que irão ser tratados e recebendo os dados tratados (ou a resposta/resultado esperado) por parte do fornecedor do serviço.

Estes serviços são definidos usando uma linguagem de descrição e possuem interfaces invocáveis, através dos quais são chamados para correr os processos que suportam. Uma vez que estes interfaces são independentes da plataforma ("*plataform-independents"*), qualquer cliente que os invoque, através de qualquer que seja o tipo de aparelho e sistema operativo que este utilize, tem a garantia que o serviço é correctamente invocado e que produz o resultado esperado.

1- Processo relativo ao uso de um *Web Service.*

Existem 3 intervenientes principais no processo de pesquisa e uso de um WS numa SOA: o Cliente ou Utilizador de serviço ("*Service Consumer*"); o

Fornecedor de serviços ("*Service Provider*"); e o Intermediário de serviço ("*Service Broker*").

O Fornecedor descreve o seu serviço usando a linguagem WSDL (*"Web Service Definition Language"* [20]), descrição essa que é publicada num *Service Broker*. O *Service Broker* irá de seguida, e utilizando a linguagem UDDI ("*Universal Description, Descovery and Integration*" [21]), anunciar/publicitar o serviço na internet. O Utilizador procura um serviço que cumpra os seus requisitos no *Service Broker*, e, depois de o encontrar, recebe deste o WSDL relativo a esse serviço, que descreve as entradas e saídas deste, de modo a que o possa utilizar e invocar (a invocação é feita usando o ficheiro WSDL) [22].

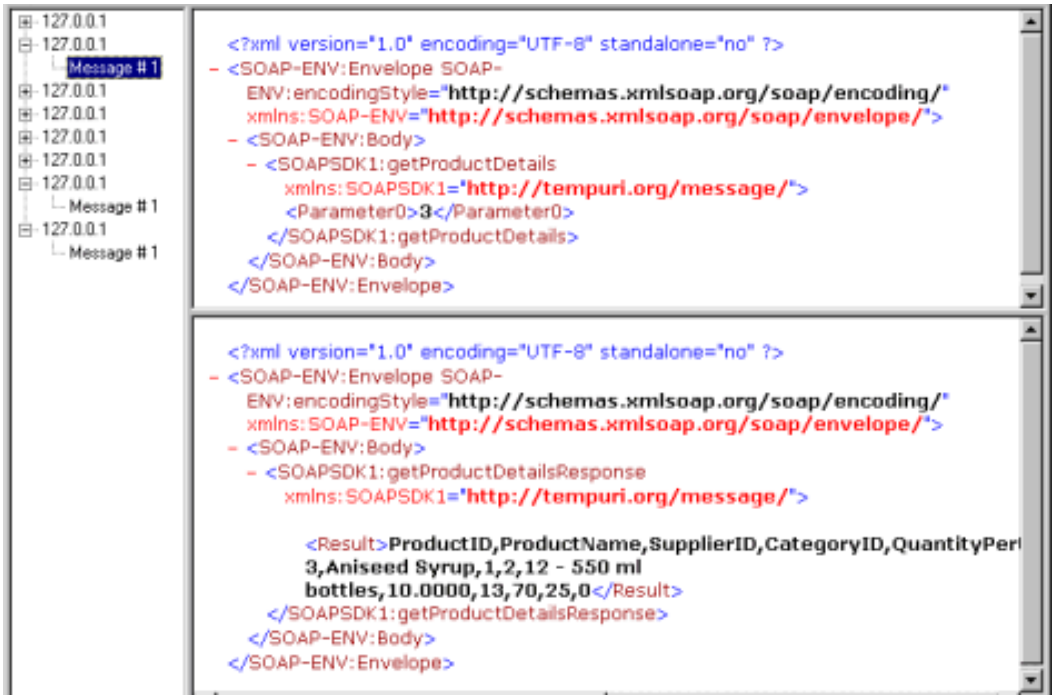
2.3 – Historia e Evolução dos Web Services

A primeira tentativa de criar um standard para a comunicação entre negócios por rede surgiu em 1975, com o lançamento do EDI ("*Electronic Data Interchange*" [23]), uma arquitectura cuja função primária passava por tratar de transacções repetitivas (encomendas, facturas, créditos, notificações, etc.) (*Turban et al*). Ao longo dos anos apareceram outras arquitecturas que tentaram ser o meio universal de condução de lógica de negócios (*business logic*) numa rede informática: CORBA ("*Common Object Request Broker Architecture*"), DCOM ("*Distributed Component Object Model*"), URPC ("*Unix Remote Procedure Call*" [24]), JRMI ("*Java Remote Method Invocation*" [25]). No entanto, nenhuma destas arquitecturas conseguiu impor-se no mercado de forma a poder ser considerada como um sucesso. Ainda existem na actualidade, e são utilizadas em alguns casos específicos, mas falharam a atingir o alcance a que se propunham, quer

devido à sua complexidade, custo, flexibilidade ou falta de suporte por parte da indústria.

Com o surgimento e posterior massificação da Web, aquilo que antes parecia impossível – que as companhias acordassem num standard para um protocolo de transporte entre serviços de redes diferentes – tornou-se não só possível como no passo lógico a tomar.

A Web corre HTTP [26] (o standard universal para negócios a desde 1997) sobre TCP/IP [27] (tecnologia já bem implementada e adulta mesmo antes da massificação da Web), e com o surgimento do XML (1998), um protocolo para encriptação de dados, independente da plataforma, que podia ser utilizado para descrever protocolos de passagem de mensagens (o que faz dele a escolha óbvia para a comunicação inter-aplicações), ficavam disponíveis assim todas as tecnologias necessárias para o surgimento dos WS, assim como o suporte necessário da indústria para que todas estas tecnologias se afirmassem, criando-se as raízes para o desenvolvimento deste conceito, que rapidamente ganhou forma e cresceu; um crescimento tão rápido que é considerado por muitos como o



2 – O SOAP é um protocolo de comunicaçao entre serviços. A figura mostra um excerto das mensagens XML trocadas entre cliente e servidor SOAP.

de maior expansão na história dos processos de negócios [28].

Finalmente, também em 1998, e baseando-se em XML, surgiu aquele que seria o protocolo base para os WS – SOAP ("*Simple Object Access Protocol*" [29]) – um protocolo criado para comunicação inter-processos flexível, geral e independente da plataforma (fig. 2), originalmente desenvolvido pela Microssoft, que viria a receber o apoio e suporte de outras grandes companhias informáticas, como a IBM (depois de alguma desconfiança inicial das empresas de IT – Tecnologias de Informação - por se tratar de um produto originário da Microssoft).

Da parceria entre a Microssoft e a IBM com vista a descrever programaticamente como conectar com WS resultou finalmente a linguagem WSDL ("*Web Services Description Language*"), que se tornaria o standard para a

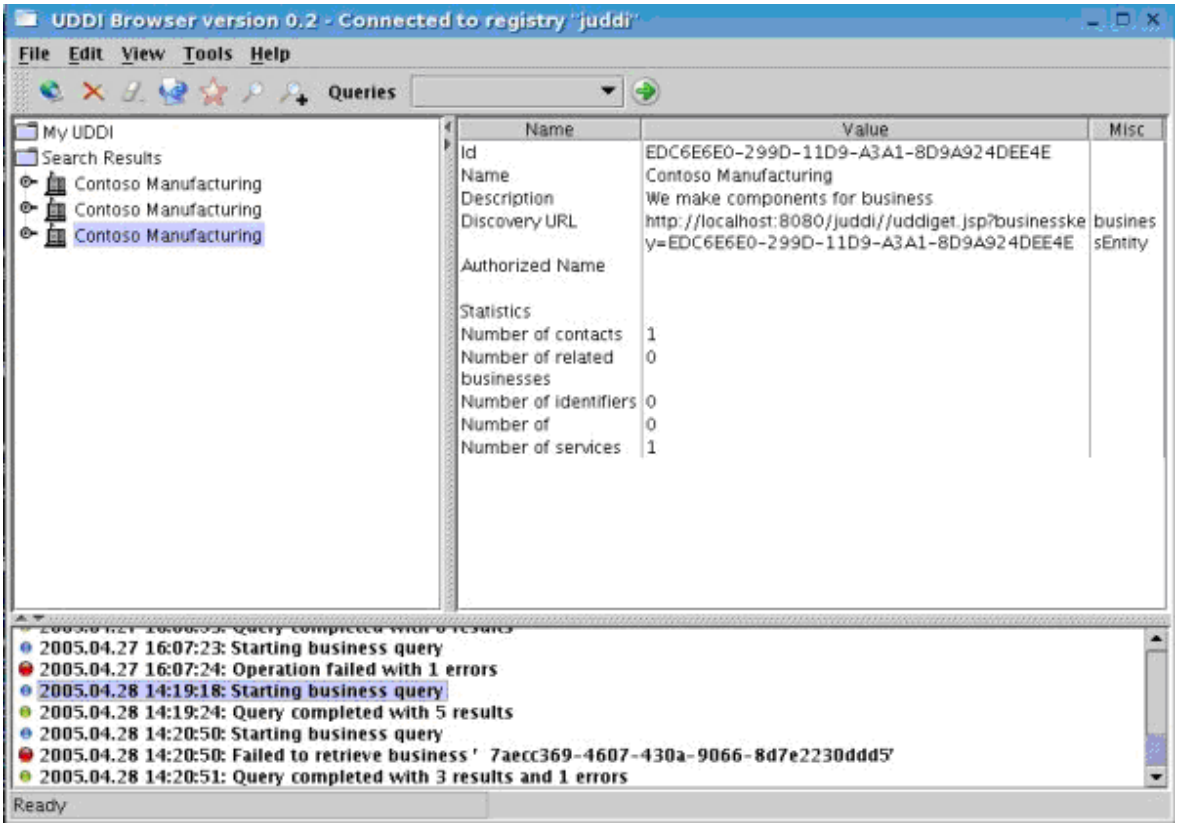
```
- <binding name="TestSoapBinding" type="tns:TestSoapPortType">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
  - <operation name="Multiply">
      <soap:operation style="rpc" soapAction="http://soapinterop.org/Multiply" />
    - <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"
          namespace="http://soapinterop.org" />
      </input>
    - <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded"
          namespace="http://soapinterop.org" />
      </output>
    </operation>
  - <operation name="Add">
      <soap:operation style="document" soapAction="http://soapinterop.org/Add" />
    - <input>
        <soap:body use="literal" />
      </input>
    - <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
- <service name="MSInterop1DocAndRPCService">
  - <port name="TestSoap" binding="tns:TestSoapBinding">
      <soap:address location="http://localhost/services/msInteropDocAndRpc" />
    </port>
  </service>
</definitions>
```

3 – A linguagem WSDL descreve o WS, os portos de entrada e saida, o tipo de dados, a localizaçao do serviço e a sua identificação.

descrição destes (fig.3).

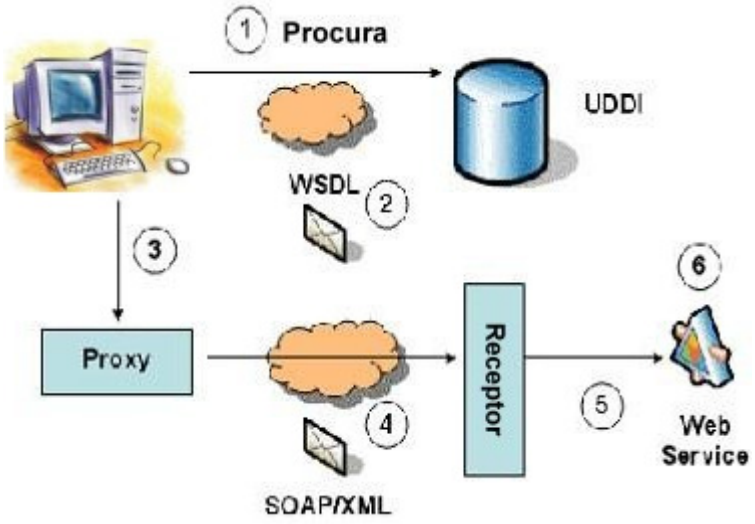
Por último, depois das companhias de software darem inicio à criação e desenvolvimento de WS, usando WSDL e SOAP, surgiu a necessidade de publicitar e procurar esses Serviços online. Para resolver esta carência, a

Microssoft, IBM e Ariba, em parceria (ano 2000), desenvolveram a UDDI ("*Universal Description, Descovery and Integration* "). No fim de 2000, com a WSDL, SOAP e UDDI a serem assumidas definitivamente como os standards para WS, foi a vez das companhias de IT darem o seu apoio formal e suporte a esta tecnologia, sendo as principais a Microssoft, IBM, Oracle, HP e Sun, um grupo tão heterogéneo como forte dentro da indústria informática, o que deu ainda mais credibilidade e força a esta tecnologia [28].



4 – Para localizar WS, são usados programas ou *browsers* de pesquisa de registos UDDI, como o "*UDDI Browser*" mostrado na figura, que fornece as informações necessárias sobre os WS que necessitamos.

De uma maneira simplista, pode-se afirmar que a UDDI diz onde os serviços estão, a WSDL descreve o que eles fazem e o SOAP fala com eles. A interacção entre estes na utilização de um WS pode ser demonstrada pelo seguinte esquema:



5 - Esquema representando os passos da utilização de um *Web Service*.

Pode-se descrever todo o processo em 6 passos [30]:

1- Procura: o utilizador efectua uma procura num site UDDI por um WS que satisfaça as suas necessidades.

2- Descrição: O site devolve um ficheiro WSDL com a descrição do serviço escolhido para a aplicação do utilizador.

3- Criação de Proxy: É criada uma proxy para o serviço remoto.

4- Criação de uma mensagem SOAP: Uma mensagem SOAP/XML é criada e enviada para a URL especificada no ficheiro WSDL.

5- Recepção: Um receptor SOAP no site alvo recebe o pedido, trata-o, interpreta-o e envia-o para o WS.

6- Função: o WS executa a sua função e retorna o resultado ao cliente, via receptor e proxy.

O que torna os WS uma tecnologia tão apetecível para os processos de negócio reside nas suas características principais, que se adequam às necessidades especificas das SOA. Temos assim a destacar as suas principais

vantagens, muitas delas comuns às SOA, uma vez que são uma implementação destas [17, 31]:

- **Encapsulação:** todos os WS são serviços, logo, o que é importante é o que faz e não como o faz. Isto reduz em muito a complexidade associada ao seu uso, uma vez que não é necessária a preocupação com os detalhes da implementação dos serviços invocados. Posterior extensibilidade destes é possível em tempo real uma vez que qualquer mudança no funcionamento interno destes é transparente para os utilizadores.

- **Abstracção:** para além do que é mencionado no contrato do serviço, toda a lógica deste é escondida ao utilizador.

- **"*Loose Coupling*":** o conceito de "*Coupling*" refere-se à dependência entre sistemas que interagem entre si. Apesar de muitos WS serem eles próprios consumidores de outros serviços ("Dependência real", que existe sempre nestes casos), os factores que estes têm que suprir para utilizar os resultados destes, como a linguagem, a plataforma, a API usadas, etc. ("Dependência Artificial"), são reduzidas ao mínimo, pois apenas os resultados importam, e estes obedecem sempre ao mesmo standard. [18]

- **Reusabilidade:** Um dos principais conceitos associados aos WS é a reusabilidade. Como os serviços são invocados remotamente, sendo todas as invocações tratadas de forma independente, possibilita a que estes sejam reutilizados sem limite por clientes ou outros serviços, sendo a única limitação o poder de processamento do servidor em que os serviços se encontram.

- **Composabilidade:** Vários WS podem ser agrupados e coordenados para formar serviços compostos.

- "***Service Statleness*":** Os serviços tentam manter o mínimo possível de informação relativa a uma actividade, isto é, não "mantêm estado" ou tentam mantê-lo ao mínimo, maximizando assim a sua escalabilidade e reduzindo o consumo de recursos, que de outro modo teriam que ser utilizados para guardar toda a informação corrente do serviço enquanto este está a ser utilizado.

- **Descoberta:** Os serviços são elaborados de modo a serem descritivos para os potenciais utilizadores, facilitando a sua descoberta e acesso através dos mecanismos de descoberta de WS.

- **Interoperabilidade com "*Legacy Aplications*":** Designam-se por *Legacy Aplications* sistemas ou aplicações antigas que ainda são usadas pelo utilizador, normalmente uma organização, por este não os querer substituir ou evoluir. Essas aplicações podem ser encapsuladas como um WS, permitindo assim que sejam utilizadas por sistemas recentes, uma vez que lhes permite interagir com estes via standards, como SOAP e HTTP.

Com todas estas vantagens, os WS tornaram-se no meio ideal de implementação das SOA, ao mesmo tempo que passaram a ser a evolução para os negócios online (*e-business*). Ao ver os sistemas segundo a perspectiva de que tudo é um serviço, dinamicamente descoberto e orquestrado através de mensagens pela rede, foi possível evoluir as técnicas orientadas ao objecto (encapsulação, passagem de mensagens, "*binding*" dinâmico e reflexão) e aplicá-las numa verdadeira SOA para o "*e-business*".

Com o evoluir das SOA, dos WS e das tecnologias a eles associadas, o conceito de Arquitecto de Software tomou forma no mundo empresarial e de informação, como um gestor, um provedor de serviços, que tem em atenção o tipo e modelo de negócio em que se insere, as suas especificações e como as soluções de IT que propõe podem adicionar valor ao negócio.

Surgiu assim a necessidade de uma linguagem, mecanismos e aplicações que permitissem ao Arquitecto de Software gerir os WS para os seus fins, isto é, de os orquestrar.

Para esse fim, foram criadas várias tecnologias distintas (apesar de na sua grande maioria serem baseadas em XML), cada uma com uma abordagem diferente para o mesmo problema – a comunicação e gestão de negócios entre si.

De entre estes destacam-se o ebXML (*electronic business* XML), um conjunto de especificações que juntas criam uma *Framework* modular para negócios electrónicos [32]; o RosettaNet [33], um standard que define as linhas para troca de mensagens, interfaces de processos de negócio e *frameworks* para a interacção de companhias por via electrónica, em especial na área das cadeias de distribuição; UBL (*Universal Business Language*), uma biblioteca de documentos de negócio electrónico, em XML, gratuita, desenhada para ser

inserida directamente em processos de negócio já existentes, providenciando um ponto de entrada a estes no comercio electrónico, eliminando a necessidade do suporte em papel cadeias de distribuição [34]; e BPEL (*Business Process Execution Language*), uma linguagem criada em especifico a pensar nos WS e na sua utilização e gestão, da qual se irá falar a seguir.

Capítulo 3 – WS-BPEL

Desenvolver WS no contexto de uma SOA implica balançar uma série de questões, como a reusabilidade e flexibilidade. No entanto, depois de ter passado a fase de design e execução do serviço e termos um "produto" pronto a ser consumido, um WS, temos que nos preocupar com uma outra questão igualmente importante – como fazer com que vários serviços distintos interajam para uma finalidade?

Esta questão levanta condições e novos requerimentos que são tratados por uma linguagem desenhada especificamente para isto, o WS-BPEL (*Web Services Business Process Execution Language*), que passarei a designar por BPEL [35].

Criada em 2003, numa parceria entre a IBM, a Microsoft e a BEA, a BPEL (inicialmente designada por BPEL4WS) foi o resultado da evolução das duas principais linguagens de "programação em grande" (programação que envolve grandes equipas de programadores, ou equipas mais pequenas, durante um período de tempo alargado, de modo a criar programas complexos de uma maneira modular e com interacções bem definidas, de modo a facilitar a sua, de outra maneira, difícil manutenção e desenvolvimento), XLANG (*eXtensible LANGuage [36]*) e WSFL (*Web Services Flow Language [37]*), da Microsoft e IBM respectivamente, que decidiram combinar ambas para fazer face às necessidades emergentes da utilização dos WS assim como para enfrentar outras linguagens que entretanto estavam a aparecer, como a BPMS.

Actualmente na sua versão 2.0 e sob a égide da OASIS (*Organization for the Advancement of Structured Information Standards,* [38]), o BPEL é considerado como o standard para a orquestração de WS, isto é, uma linguagem usada para programar processos de negócios; tendo o suporte de várias dezenas de companhias (de entre elas destacam-se Adobe Systems, Avaya, BEA Systems, Booz Allen Hamilton, Electronic Data Systems, Hewlett-Packard, NEC, Novell, Oracle, etc.), que estão a traçar o rumo que este vai tomar.

Assim, numa perspectiva cronológica da evolução do BPEL até à sua versão actual, temos:

- **WSFL**, Maio 2001 (IBM) - *Web Services Flow Language*
- **XLANG,** Maio 2001 (Microsoft) – O nome não representa um acrónimo, apesar de ser escrito em letras maiúsculas, sendo uma abreviatura de *eXtreme LANGuage.*
- **BPEL 1.0**, Julho 2002 (BEA, IBM, Microsoft) – A fusão entre WSFL e XLANG.
- **BPEL4WS 1.1**, Março 2003 (BEA, IBM, Microsoft, SAP, Siebel) – A especificação submetida à OASIS.
- **WS-BPEL 2.0**, Março 2007 (OASIS; 39 outras companhias membros do comité técnico) – A primeira versão como um standard aprovado por uma organização de standards, e ainda a versão actual.

BPEL usa uma linguagem baseada em XML, com suporte para o *stack* dos *WS*, incluindo WSDL, SOAP, UDDI, mensagens compatíveis com WS, endereçamento, coordenação e transacção de WS [39], existindo já várias implementações gráficas de motores BPEL, o que permite que a sua utilização seja mais acessível e prática.

Dentro de uma empresa, o uso do BPEL é indicado para a normalizar a integração das suas aplicações, assim como integrar sistemas que de outro modo estariam isolados. Entre empresas, permite uma integração entre parceiros de negócios mais fácil, estimulando-as a aprofundar a definição dos seus processos de negócio, optimizando assim a sua organização e a escolha dos mais adequados às suas necessidades [40].

Os processos de negócios definidos em BPEL não afectam os sistemas existentes, o que vai estimular o upgrade destes. Tudo isto faz desta linguagem uma tecnologia chave para ambientes em que as funcionalidades são ou irão ser expostas via WS, aumentando a sua importância proporcionalmente ao aumento da utilização dos WS [39].

Apesar de ser possível realizar o que é feito através do BPEL com linguagens usuais, como java, o resultado seria confuso e complicado de realizar. No entanto, usando BPEL, que fornece uma camada de abstracção superior, assim como interfaces gráficos que permitem a visualização dos processos de negócio que o utilizador pretende implementar, todos os passos da execução podem ser ligados e geridos mais facilmente.

Na prática, um processo em BPEL especifica a ordem em que cada WS participante deve ser invocado, em série ou em paralelo, permitindo invocações condicionais (por exemplo, quando um WS necessita do resultado de um outro para poder cumprir a sua função). Suporta ciclos, declaração de variáveis, cópia e atribuição de valores, detecção e tratamento de falhas, etc. Deste modo, processos de negócio complexos são descritos em grafos, sendo tratados de um modo algorítmico.

Um cenário típico de utilização envolve a recepção de um pedido, *request*, pelo processo de negócio escrito em BPEL; o processo invoca (operação *invoke*) os WS envolvidos, respondendo de seguida para a aplicação que o chamou (*reply*). Uma vez que um processo em BPEL comunica muito com outros WS, toda a sua execução se apoia muito na descrição destes em WSDL [16].

3.1 – Orquestração e Coreografia

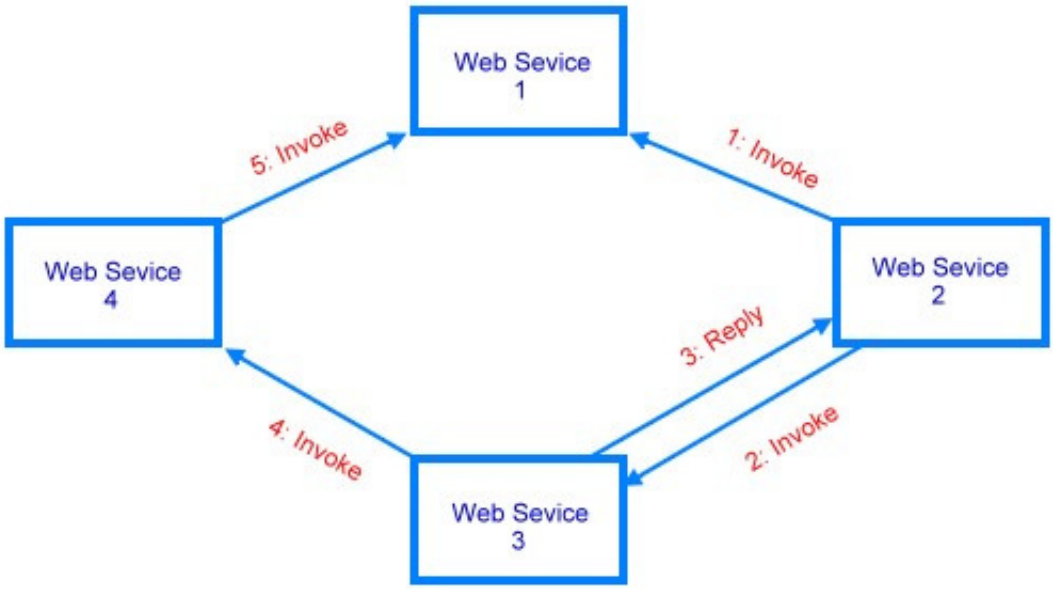
Os analistas são bastante consensuais ao descrever o BPEL como o standard para “orquestrar” processos de negócios, o que significa que o controlo centralizado de WS passará por um motor BPEL [39]. É de notar também que este protocolo deixa em aberto a possibilidade de “coreografar” WS, isto é, gerir e interagir WS sem uma unidade central a controlar o processo. A habilidade do BPEL em publicar os requerimentos de execução de um WS pode ser o necessário para que a visão mais descentralizada, a coreografia, seja possível.

Tem-se assim dois tipos de combinação possível de WS, a Orquestração e a Coreografia.

A Orquestração, mais usual em processos de negócios privados, consiste em ter um processo central, normalmente um WS, que controla e coordena as diferentes operações de todos os outros WS envolvidos na operação. Deste modo, apenas o WS central tem "consciência" do objectivo global que todos desempenham, sendo o responsável por definir explicitamente as operações e a ordem de invocação dos WS envolvidos, ao passo que estes não sabem, nem precisam saber, que estão a contribuir para um projecto maior.

6 - Composição de *Web Services* com orquestração

Coreografia, por outro lado, não necessita de um coordenador. Isto porque cada WS tem consciência do seu papel no processo geral – sabe quais são os WS com os quais interage e quando executar as suas operações. Coreografia é assim um esforço coordenado de vários WS, tendo por base a troca de mensagens em processos de negócio públicos. Assim, todos os participantes de uma coreografia têm consciência do processo de negócio, das operações a executar, das mensagens a trocar, tal como o *timing* destas.



7 - Composição de *Web Services* com coreografia.

Destas duas abordagens, a orquestração apresenta vantagens em relação à coreografia no que respeita à composição de WS para executar processos de negócios, uma vez que é uma abordagem mais flexível:

- A coordenação dos processos é feita por um coordenador conhecido.

- Os WS são incorporados sem necessitarem de ter consciência que pertencem a um processo de negócio maior.

- Em caso de falha, um cenário alternativo pode ser rapidamente posto em prática pelo coordenador.

O BPEL suporta duas maneiras de descrever processos de negócio, dependendo se o objectivo é orquestrar ou coreografar. Assim, “Processos Executáveis” permitem especificar os detalhes de processos de negócio; seguem a Orquestração e são executados por um motor de orquestração. “Protocolos de Negócio Abstractos” apenas permitem especificar a troca de mensagens públicas entre as partes, não incluindo os detalhes dos processos internos e sua execução, não sendo também executáveis; seguem portanto o paradigma da Coreografia.

3.2 – A linguagem BPEL

Como foi referido anteriormente, o BPEL é uma linguagem baseada em XML vocacionada para o tratamento, invocação e manipulação de WS.

O código fonte de um processo de negócio nesta linguagem apresenta uma estrutura reminiscente do XML, com cada função e componente (denominados por "actividades" no contexto do BPEL) escrito sequencialmente, sendo cada uma identificada pela expressão "`bpel:`", que a identifica como uma expressão desta linguagem, e não XML normal.

Cada linha de código é delimitada pelos sinais de menor e maior, "<" e ">". Caso a linha de código seja única, isto é, não tenha um domínio associado, possui um carácter que marca o final, "/", antes do sinal de ">" (`<bpel:link name="L1"/>`); caso possua um domínio, a linha inicial possui o tipo de actividade a que se refere, limitada pelos sinais normais (`<bpel:sources>`), sendo a linha final do domínio identificada com a mesma expressão da primeira, mas precedida pelo carácter "/" (`</bpel:sources>`). Tudo o que se encontra entre estas duas expressões pertence ao seu domínio.

```
<bpel:sequence>
   <bpel:assign name="IncrementTotal">
      <bpel:copy>
         <bpel:from>number( $orderMessage.order/OrderDetail[ $detailIndex ]/QTY )</bpel:from>
         <bpel:to variable="itemQuantity"/>
      </bpel:copy>
      <bpel:copy>
         <bpel:from>number( $orderMessage.order/OrderDetail[ $detailIndex ]/Cost )</bpel:from>
         <bpel:to variable="itemCost"/>
      </bpel:copy>
      <bpel:copy>
         <bpel:from>($orderTotal + ($itemQuantity * $itemCost))</bpel:from>
         <bpel:to variable="orderTotal"/>
      </bpel:copy>
   </bpel:assign>
   <bpel:assign name="IncrementDetailIndex">
      <bpel:copy>
         <bpel:from>($detailIndex + 1)</bpel:from>
         <bpel:to variable="detailIndex"/>
      </bpel:copy>
   </bpel:assign>
</bpel:sequence>
```

8 – Exemplo de uma actividade escrita em BPEL, uma actividade de *sequence*, com várias actividades de *assign* dentro do seu domínio

Um processo de negócio escrito na linguagem BPEL é assim conjunto de actividades ordenadas que começa por uma instrução de *receive* ou *pick* (uma vez que, sendo o processo de negócio ele mesmo um WS, a sua execução só começa quando é invocado).

O BPEL possui uma sintaxe com vários tipos de actividades, cuja conjunção consegue descrever todo o tipo de processo de negócio que um utilizador necessite. Na tabela seguinte são mostradas essas actividades, assim como uma pequena descrição destas.

Instrução	**Acção**
<invoke>	Chama um serviço externo de forma síncrona.
<empty>	É uma actividade "sem operação", útil para sincronizar actividades concorrentes.
<assign>	Manipular variáveis de dados e referências de destino de *partner links*, criando operações de cópia (*copy to/from*) para estes.
<suspend>	Suspende um processo em execução. Útil numa actividade de *catch* ou *catchAll* para "apanhar" erros inesperados.
<receive>	Recebe entradas de serviços assíncronos.
<reply>	Envia respostas de serviços assíncronos
<wait> <until>..	Pausa o fluxo do processo de negócio durante um determinado período de tempo.
<exit>	Pára um processo de negócio.
<validate>	Valida os valores de variáveis em relação à concordância com o tipo de dados.
<continue>	Termina a iteração do ciclo (*while*, *forEach*, *repeatUntil*) em que foi usada e passa para a próxima.
<break>	Faz com que o ciclo actual (*if*, *while*, *forEach*) termine.
<throw>	Um meio de lidar com falhas durante o a execução do processo, sinalizando as que ocorrem no seu *scope*.
<rethrow>	Passa para o *scope* em que está inserida as falhas que são "apanhadas" pelo *fault handler* local.

<compensate>	Compensa todos os processos internos de um *scope* que já terminaram, quando num deles ocorre um erro, revertendo essas actividades.
<compensateScope>	Faz a compensação dos processos internos de um *scope* alvo.
<opaque>	Apenas utilizada em processos abstractos (não executáveis), substituindo nestes uma actividade que seria utilizada num processo executável.
<sequence>	Permite definir uma série de actividades que serão invocadas numa sequência ordenada.
<flow>	Executa todas as actividades dentro do seu *scope* em paralelo.
<if>	Executa uma actividade baseando-se numa ou mais condições definidas no elemento "*if*" e nos elementos adicionais "*else if*" e "else". As condições são analisadas por ordem, sendo que a primeira que se verificar terá a actividade correspondente executada
<while>	Executa uma actividade repetidamente, enquanto a condição que lhe está associada for verdadeira.
<pick>	Contém pelo menos uma mensagem ou parte de mensagem e um ou mais alarmes. Quando é executada, esta actividade bloqueia o processo até que a mensagem seja recebida ou, caso contrário, um dos alarmes execute.
<scope>	Providencia um contexto para um conjunto de actividades.
<repeatUntil>	Executa uma actividade repetidamente até que a sua condição seja verdadeira.
<forEach>	As actividades contidas no seu *scope* são executadas um determinado número de vezes, em sequência ou em paralelo. O número de vezes é dado por uma expressão, que vai correr N iterações, em que N varia de um valor inicial a um valor final.
<elseif> <else>	Definem condições extra num ciclo *if*. Podem existir vários *elseIf* por ciclo, e um *else*, normalmente usado como a escolha por defeito.
<catch>	Um *fault handler*, que define no seu scope as actividades a

	executar caso uma falha com um nome e/ou tipo especifico ocorra.
<catchAll>	Um *fault handler* que executa quando uma actividade de *throw* não é apanhada por nenhum *catch.*
<onEvent>	Indica que o evento especificado no seu *scope* está à espera de uma mensagem para ser despoletado, quer esta seja de uma actividade *receive/reply* ou operação de sentido único (*one-way*).
<onAlarm>	Define um evento temporal de *timeout*, quer através de uma duração, quer através de um limite (*deadline*), despoletando um alarme quando o *timeout* ocorre.
<onMessage>	Define um conjunto de actividades a executar aquando da recepção de uma mensagem.

Tabela 1 – Resumo das actividades em BPEL.

Assim, temos as actividades de interacção com WS, das quais o *receive* faz parte, em que é recebida uma mensagem com dados de um serviço "colaborador" (*service partner*) com o qual esta a comunicar. Opcionalmente, pode começar um processo, como referido acima, criando uma instância deste.

As outras actividades de interacção com WS são o *invoke*, que invoca WS externos, e o *reply*, que retorna uma resposta para um serviço identificado no *receive* correspondente (formando um par *receive*/*reply*). A instrução de *invoke* especifica qual o *partner link* (uma construção BPEL que descreve os papeis que um processo e um serviço desempenham) e a operação que vai ser realizada, assim com a variável de entrada para a mensagem de dados e de saída (no caso da invocação de um WS de pedido - resposta).

Para além das actividades de interacção, existem as actividades de processamento interno, que servem para construir um processo de negócio, um fluxo de acção à volta dos WS de modo a obter-se o efeito desejado.

Este é o tipo de actividades mais numeroso, abrangendo as actividades de ligação, de ciclo, de atribuição e de tratamento de erros, etc., sendo enumeradas de seguida:

Validate é a actividade responsável por validar valores de variáveis, isto é, comparar valores atribuídos com os tipos de valores esperados para variáveis, de

modo a prevenir conflito de dados. Tem como argumentos as variáveis que queremos validar.

Um *assign* faz a actualização do conteúdo de variáveis. Essa actualização pode ser feita de diversas maneiras: através da cópia de valores entre variáveis; construir novos dados usando funções e expressões de Xpath (ou outra linguagem); usando extensões BPEL para construir novos dados. Outra das utilizações usuais desta actividade consiste em copiar referências de destino de e para *partner links*, de modo a permitir a selecção dinâmica de serviços "colaboradores". Possui como argumentos a origem da cópia (designada por "*from*") e o destino da cópia (designada por "*to*"). A origem pode ser de vários tipos, desde *literal* (que designa uma expressão, uma frase, isto é, algo constante e predefinido pelo programador), a uma expressão (normalmente em Xpath), variável, opaca, *partner link* ou uma propriedade de uma variável. A cada um destes tipos de origem, corresponde um tipo de destino específico, que tem que ser tido em conta de modo a evitar conflitos de dados/atribuições.

Uma actividade de *throw* é a mais simples das actividades de tratamento de erros, sinalizando uma falha quando esta ocorre e especificando qual o tipo dessa falha. O argumento obrigatório é o nome da falha que se pretende apanhar.

Outra das actividades de tratamento de erros é a *rethrow*. Esta passa uma falha "apanhada" pelo *fault handler* (gestor de falhas, que possui já uma actividade de *throw*) local para o *scope* (domínio) em que está inserido ("Domínio Pai"). Este domínio pertence a uma actividade de *catch* ou *catchAll*, que vai receber as falhas que ocorram, para de seguida as tratar.

Para terminar imediatamente um processo executável, existe a actividade *exit*, que faz com que todas as actividades ainda em execução sejam terminadas, sem que ocorram erros nem haja compensação.

Um *wait* faz com que um processo pare por um determinado período de tempo ou até que uma hora limite seja alcançada. Os seus argumentos são uma o tipo de espera que irá ser feita – um intervalo temporal ou um tempo limite – e uma expressão que defina qual a duração ou limite pretendido.

Um pouco semelhante à anterior é a actividade *suspend*, que suspende um processo em execução, de modo a permitir que falhas inesperadas sejam apanhadas por uma actividade de *catch* ou *catchAll*, permitindo que estas sejam revistas manualmente pelo utilizador.

Empty é uma actividade vazia, que não faz nada quando executa. É usada para sincronizar actividades concorrentes, assim como em alturas que não se quer que seja feita qualquer acção.

Compensate Scope é uma actividade usada em gestores de falhas, de modo a fazer a compensação de uma parte do processo que já tenha terminado aquando da ocorrência de um erro. Deste modo, se algum erro ocorrer dentro do domínio em que está inserido, esta actividade vai desfazer o que já tinha sido feito com sucesso nesse domínio, de modo a que não ocorram situações de partes de processos ou mesmo processos inteiros parcialmente concluídos. Um exemplo disto é um caso em que se quer reservar um quarto e um fato entregue nesse quarto; se ocorrer um erro na reserva do quarto, mas anteriormente a reserva do facto foi feita com sucesso, é preciso desmarcar essa reserva, uma vez que já não esta correcta. *Compensate scope* é assim uma actividade extremamente útil, em especial em processos com uma complexidade maior.

Na mesma linha que a actividade anterior, existe o *compensate*, que faz o mesmo que esta, mas sem um domínio (*scope*) associado. Esta actividade pode ser associada a qualquer domínio que seja definido como parâmetro ou, caso nenhum seja, vai fazer a compensação geral do processo de negócio.

Dentro de actividades cíclicas, como *While, forEach* ou *repeatUntil*, são encontradas duas actividades: *break* e *continue*. Um *continue* termina a iteração do ciclo em que se encontra, passando para a próxima, de acordo com as regras específicas de cada tipo de ciclo. Um *break* vai terminar não só a iteração em que ocorreu, mas todo o ciclo, terminando também todas as iterações que ainda estivessem a decorrer em paralelo.

A actividade *opaque* apenas é usada em processos abstractos (não executáveis), sendo usada para substituir uma actividade que pertenceria a um processo executável.

Finalmente, aparecem as actividades estruturantes do BPEL. Estas englobam os ciclos, já referidos anteriormente, assim como actividades de fluxo e sequência e actividades condicionais.

As actividades que permitem a execução de ciclos são três: *Repeat Until*, *While* e *For Each*, cada uma trata as condições de maneira diferente, permitindo a escolha da mais apropriada para cada situação.

Repeat until vai executar uma actividade repetidamente até que a sua condição seja verdadeira, contrariamente à actividade *while*, em que o ciclo executa enquanto a condição for verdadeira, terminando quando esta for falsa. Ao contrário do *while*, num ciclo *repeat until* a actividade é executada pelo menos uma vez, já que a condição é verificada apenas no final do ciclo, ao contrário do *while*, em que só depois de verificada a condição é executada a primeira iteração do ciclo.

A terceira actividade de ciclo é a *for each*, que funciona um pouco como as instruções "*for*" de linguagens como o C ou C++, isto é, dentro de um intervalo de ocorrências, definido por um valor inicial e final de uma variável, sempre um valor inteiro positivo, e com um incremento de uma unidade, são executadas N iterações de uma actividade, em que N é a diferença entre os valores iniciais e finais. As actividades podem ser executadas em paralelo (todas ao mesmo tempo) ou sequencialmente, o que torna esta actividade moldável para vários tipos de situações, sendo possível também a utilização de um contador com o número de iterações completas. Os argumentos desta actividade consistem no nome do contador, no seu valor inicial e final, e de uma *flag* com a opção da execução ser ou não em paralelo (por defeito a execução é sequencial).

A actividade *if* é designada como uma actividade condicional, uma vez que executa um conjunto de actividades apenas se alguma das suas condições se verificar (cada uma descrita num ramo *if*, no caso da primeira, e *else if* no caso de todas as outras), caso contrário executa a sua condição *else*, se existir, ou nenhuma acção, caso não possua um ramo *else*.

A actividade de *pick* contém pelo menos uma mensagem ou parte de mensagem e um ou mais alarmes, sendo que quando uma actividade de *pick* executa, o processo é bloqueado até que receba uma das mensagens ou um dos

seus alarmes dispare (um alarme também é uma actividade BPEL, designada por *onAlarm*). Esta actividade também pode ser usada para iniciar um processo de negócio, com um conjunto de mensagens e sem alarme. Para tal, uma das propriedades desta, *Create Instance*, tem que estar activa (isto é, o seu valor tem que ser *yes*), sendo executada quando a primeira mensagem é recebida.

A actividade de *scope* providencia um contexto para um conjunto de actividades, englobando num domínio uma unidade lógica de trabalho de um processo, e permitindo que essa unidade seja gerida como um todo.

Por último, existem as actividades de controlo de fluxo, *flow* e *sequence*. *Sequence* é um contentor que executa as actividades nele contidas de um modo sequencial, isto é, cada actividade só começa a ser executada quando a anterior terminar a sua execução. *Flow* é também ela um contentor, só que, ao contrário da actividade de *sequence*, faz com que todas as actividades nela contidas sejam executadas em paralelo, isto é, todas elas começam a sua execução simultaneamente, sendo que o *flow* só termina quando todas as suas actividades completam com sucesso.

Estas são as descrições das principais actividades presentes num projecto BPEL. Para além dos parâmetros obrigatórios referidos, possuem também parâmetros/propriedades opcionais, que lhes permitem ser mais versáteis e específicas. Cada uma delas pode ser usada em diferentes situações, para a obtenção de diferentes resultados, dando um grande controlo ao programador sobre o processo de negócio que quer construir.

Capítulo 4 – Motores e aplicações para BPEL

Existem já no mercado várias soluções de orquestração de WS, baseadas em BPEL, isto é, aplicações que permitem a utilizadores, quer particulares quer empresariais, desenvolver processos de negócio tendo por base esta linguagem, sendo possível também fazer o *deploy* destes na Web, para posterior invocação.

Estas aplicações e "motores" BPEL (BPEL *Engines*) passam por *Plugins* de outras ferramentas de edição de código já existentes, assim como aplicações criadas especificamente com vista ao design de processos em BPEL, existindo várias destas ferramentas disponibilizadas gratuitamente, para utilizadores particulares, sendo muitas delas também *open source*, o que permite que sofram evoluções regulares por parte da comunidade de utilizadores; assim como versões pagas, mais completas e com melhor suporte, especialmente desenhadas para processos numa escala maior, normalmente empresarial.

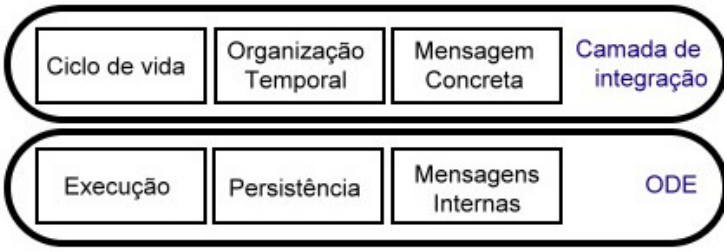
4.1 – Apache ODE

O primeiro motor BPEL estudado no âmbito desta dissertação, o ODE (*Orquestration Director Engine*) foi desenvolvido pela Apache *Foundation*, tendo evoluído da sua fase de incubação/desenvolvimento em 2007 para se tornar num dos seus principais projectos. Neste momento encontra-se na versão 1.2, lançada em Julho de 2008, que veio corrigir alguns *bugs* e problemas da primeira versão, assim como adicionar novas funcionalidades.

A filosofia do ODE em relação ao BPEL é a de que esta é a linguagem para descrever como implementar capacidades de comunicação baseada em mensagens entre serviços, não só em relação a processos de negócio, mas a todo o tipo de actividades envolvendo WS. Quer assim implementar uma

orquestração sem recorrer a interfaces gráficos (GUI, *Graphic User Interfaces*), ambientes integrados para desenvolvimento do software (IDE, *Integrated Development Environment*) ou outras bases que não apenas algum uso de XML, encapsulando detalhes de concorrência, continuação durável, fiabilidade e recuperação de falhas. Ao ser disponibilizado como um componente ao invés de um sistema (*framework*), o ODE tenta impulsionar os programadores a introduzir orquestração nos seus sistemas [41].

O ODE, no seu núcleo (*core*), usa uma implementação baseada numa camada de integração para receber e enviar mensagens para serviços externos e para aceder a recursos, como threads. Essa camada possui implementações quer para AXIS2, em que os processos do ODE são expostos como WS, quer para *Service Mix*, em que essa exposição é feita como um motor de serviço JBI (*Java Business Integration*) .



9- Camada de integração do ODE

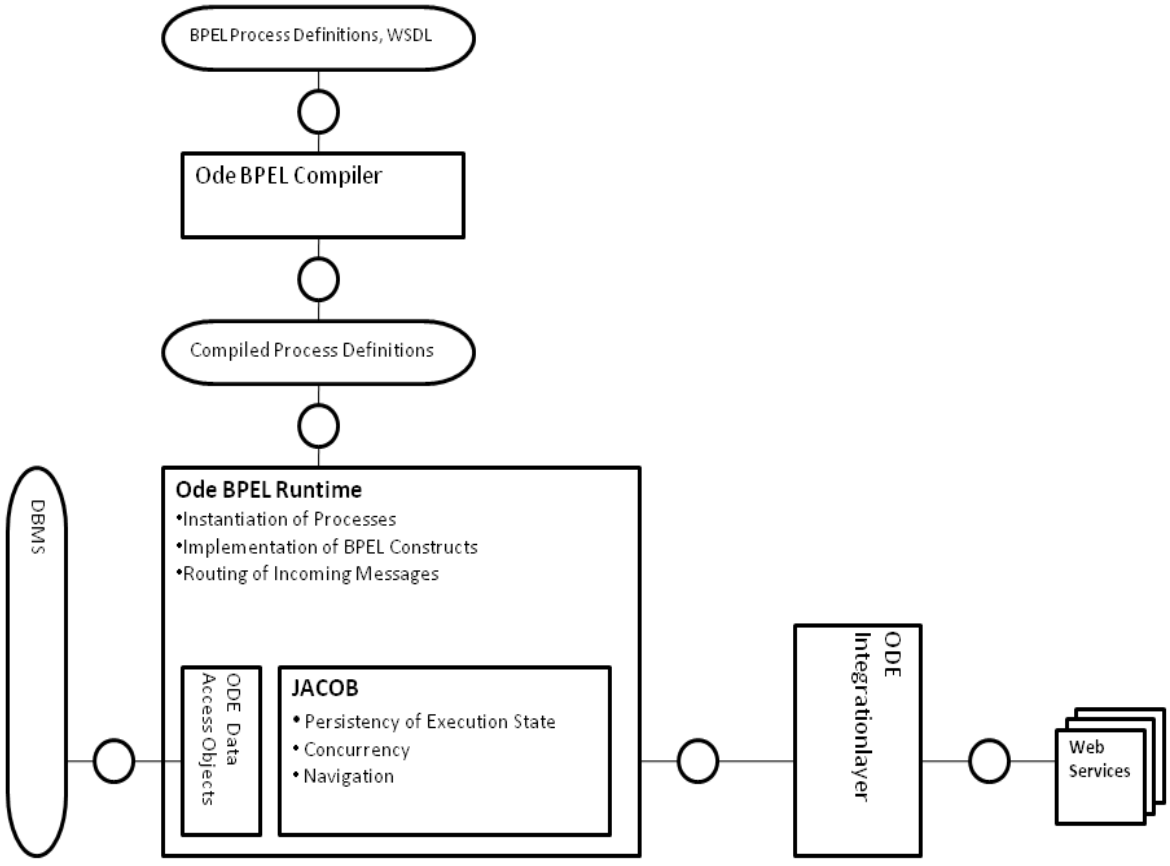
Dois dos aspectos mais desafiantes para implementar um motor BPEL envolvem a representação do estado de um processo em execução e a gestão da concorrência de processos. Para suprir estes aspectos, o ODE utiliza o Jacob (*Java Concurrent Objects*), uma *Framework* que permite representar o estado dos processos como uma colecção de objectos, ligados por canais de transmissão de mensagens; não mensagens "pesadas" de WSDL como as enviadas e recebidas externamente, mas simples mensagens internas Java, muito mais leves. Ao receber uma mensagem, um objecto pode criar novos canais e objectos e enviar mensagens. Assim, a concorrência é gerida através do consumo de mensagens num processo *single-thread* por cada par processo/instância [41]. Essencialmente, o Jacob fornece uma máquina virtual persistente para executar código/construções em BPEL.

Sendo um motor BPEL, o ODE pode ser instalado em dois ambientes distintos, sendo que ambas as instalações possuem uma base de dados própria embutida (Derby):

- Como um WS em AXIS2 (um "motor de núcleo" – *core engine* – de WS da Apache), cuja implementação (*deploy*) é feita usando um servidor de aplicações (sendo o mais comum o Tomcat) e invocado usando SOAP/http.

- Como um conjunto de serviços JBI (*Java Business Integration*), que podem ser implementados através de um repositório JBI (por exemplo, ServiceMix).

Olhando para o esquema da arquitectura do ODE, podem-se distinguir os seus vários componentes principais: o Compilador BPEL (*ODE BPEL Compiler*), o



10- Arquitectura do ODE

(http://ode.apache.org/architectural-overview.html).

motor de execução BPEL (*ODE BPEL Runtime Engine*), ODE DAOs (*Data Access Objects*), camadas de integração (*ODE Integration Layers*) e ferramentas dos utilizadores.

O compilador é o responsável, como o nome indica, por compilar o código base de BPEL (ficheiros de descrição dos processos, WSDL e esquemas), isto é, converter todos os elementos associados ao BPEL em código executável, sendo o seu output um ficheiro com o código compilado correctamente (normalmente com a extensão ".cbp") em caso de sucesso ou uma lista de erros encontrados durante a compilação caso contrário.

O motor de execução, *runtime*, recebe o ficheiro resultante da compilação descrita acima, sendo o responsável pela execução do código compilado. O *runtime* fornece as implementações para as várias estruturas do BPEL, assim como implementa toda a lógica necessária para determinar quando uma nova instância é necessária e para que instância se deve encaminhar cada mensagem recebida. Também a aplicação de gestão de processos (*Process Management API*) é implementada pelo *runtime*, sendo utilizada pelas ferramentas do utilizador para interagir com o motor.

De modo a garantir viabilidade num ambiente não viável, o *runtime* apoia-se na DAOs para garantir a persistência dos processos, normalmente via a utilização de uma base de dados relacional transaccional.

Ao nível das instâncias, o *runtime* implementa as construções BPEL via Jacob, que proporciona um mecanismo de concorrência ao nível das aplicações, sem depender de *threads*, assim como um mecanismo transparente para interromper a execução, enquanto garante a manutenção do estado.

O ODE *Data Access Objects,* DAOs, serve de mediador entre o *runtime* e uma camada inferior de dados armazenados, tipicamente uma base de dados relacional JDBC (*Java Database Connectivity*), sendo o DAOs implementado via uma biblioteca de acesso a dados OpenJPA (*Java Persistence API*). Assim, *runtime engine* requer que o DAOs resolva os seguintes problemas de persistência:

- Instâncias activas: manter um registo actualizado das instâncias criadas.
- *Routing* de mensagens: qual instância está à espera de qual mensagem.
- Variáveis: o valor das variáveis BPEL de cada instância.
-*Partner Links*: os valores dos *partner links* BPEL em cada instância.
- Estado da execução dos processos: o estado serializado da "máquina de persistência virtual" (*virtual persistence machine*) do Jacob.

As camadas de integração resolvem a necessidade do motor de execução tem em interagir com o mundo exterior, uma vez que é incapaz por si mesmo.

Assim, o que estas fazem é envolver o *runtime* num ambiente de execução, fornecendo-lhe canais de comunicação, assim como mecanismos de organização de *threads*, ao mesmo tempo que gere o seu ciclo de vida (isto é, configura e inicializa o *runtime*).

Para criar os processos em BPEL, utilizam-se aplicações, gráficas ou textuais que permitam codificar os processos. Uma das mais utilizadas para este efeito é a ferramenta "Eclipse", um IDE (*Integrated Development Environment*, ou ambiente de desenvolvimento integrado) com um interface gráfico que suporta as actividades e componentes do BPEL [Apêndice 1].

4.2 - Netbeans

Desenvolvido pela "Sun Microsystems", o Netbeans é um IDE (*Integrated Development Environment* – Ambiente de desenvolvimento integrado) java, gratuito, multi-plataformas e de código aberto para desenvolvimento de *sofware*, sendo, a par do Eclipse, um dos IDEs com mais sucesso e mais utilizado em todo

o mundo, com um grande número de contribuidores que aperfeiçoam e criam novas funcionalidades para ele, estando por isso em constante crescimento e desenvolvimento.

Nativamente, o Netbeans suporta as linguagens C++, Java, EJB e Ruby, entre outras, possuindo, desde a sua versão 5.5, um editor gráfico para BPEL [Apêndice 2]. Ao criarmos um projecto BPEL em Netbeans, este cria vários ficheiros: um ficheiro ".xsd", onde são definidos os tipos de variáveis usadas no projecto (não sendo obrigatório para o seu funcionamento, este ficheiro é útil na definição de tipos de variáveis próprios do utilizador); um ficheiro ".wsdl", em que são descritos as propriedades e funcionalidades do programa, assim como as operações definidas neste; finalmente, um ficheiro ".bpel", em que é feita a implementação de todas as operações e do fluxo do programa, isto é, é neste ficheiro que o utilizador vai "programar", sendo que possui a aparência de uma folha em branco, onde são largados os elementos e as ligações que vão constituir a actividade, num sistema "*drag-and-drop*"(clicar num elemento e arrastá-lo para a área de trabalho do Netbeans), estando todos os elementos disponíveis numa palete lateral.

Os projectos BPEL desenvolvidos, que definem um processo de negócio, em Netbeans, são adicionados a um projecto do tipo *Composite Application* de modo a poder ser feito o seu *deploy* e teste. Para que a *Composite Application* seja executada (*deployed*) no motor de serviço BPEL, é necessário que esta inclua um módulo JBI criado a partir do módulo de BPEL, isto é, um projecto nesta linguagem é tratado como uma JBI pelo Netbeans [42].

4.3 – ActiveBPEL

Consistindo num motor BPEL e num IDE, o ActiveBPEL é uma aplicação *standalone*, *open source* e sob a licença da Active Endpoints, que pretende englobar numa só aplicação todos os standards de WS, baseando-se em BPEL.

O activeBPEL pode considerar-se com a versão freeware, de desenvolvimento e não empresarial de um outro produto desta companhia, o activeVOS. Este é um produto assumidamente empresarial, possuindo algumas funcionalidades próprias para projectos de grandes dimensões e de maior complexidade. Com uma licença paga e com um maior suporte, assim como um grande cuidado no teste das aplicações e soluções para encontrar falhas e erros e consequente recuperação, o activeVOS garante o funcionamento de processos críticos para uma empresa. Apesar disso, o activeBPEL apresenta todas as funcionalidades e ferramentas para desenvolver qualquer projecto em BPEL que um utilizador particular ou pequena empresa pretenda.

Apesar de não ser *open source*, como o motor BPEL, o IDE [Apêndice 3] é disponibilizado pela Activepoints para download gratuito no seu site (http://www.activebpel.org/products/activebpeldes/get_code.html)

O activeBPEL designer possui um *servlet* incorporado, um *subset* do conhecido Apache Tomcat, plenamente funcional. Deste modo, não é necessária a instalação de um servidor externo para fazer o *deploy* dos processos criados nesta ferramenta. Para além de designer, o activeBPEL possui também incorporado um motor BPEL, pelo que fornece todas as ferramentas numa só aplicação para desenvolver, testar e executar programas escritos em BPEL [43].

Para fazer o *deploy* dos processos BPEL, o motor activeBPEL "empacota" todos os seus componentes num ficheiro ".bpr" (*business process archive*), que é de seguida automaticamente executado, existindo também a possibilidade de gerar o *script* Ant (um pouco como o *make* de programas em Linux, mas baseado na linguagem Java) para posterior re-execução do ficheiro.

Durante a criação do ficheiro "bpr", é gerado também um catálogo de recursos, automaticamente, pelo assistente bpr. A informação desse catálogo é

mostrada na consola de administração do motor, providenciando também um meio deste procurar ficheiros WSDL, esquemas e outros recursos no arquivo de execução.

Todos os projectos construídos, depois de feito o seu *deploy*, podem ser encontrados na consola de administração [Apêndice 4], onde podem ser inspeccionados enquanto "correm", sendo que é possível visualizar os detalhes do processo assim como o seu diagrama. Para processos de grandes dimensões, é fornecido também um filtro que permite seleccionar um processo em específico para visualizar.

4.4 – Análise comparativa dos motores

Para além dos motores e designers descritos anteriormente, existem mais escolhas de implementação e desenvolvimento de processos de negócio em BPEL. Mas qual destas será a melhor para um utilizador aprender a linguagem e familiarizar-se com o conceito?

Debruçando apenas sobre os 3 motores descritos, ActiveBPEL, Netbeans e Apache ODE, e da análise da documentação disponível, suporte, qualidade dos designers, poder do motor BPEL e facilidade de *deploy* dos projectos de cada um destes, pode-se afirmar que o ActiveBPEL leva vantagem em relação aos restantes.

Um dos factores é o facto de ser uma ferramenta completa, com um designer, um motor BPEL e um servidor (Tomcat) incluídos na mesma instalação facilita o *deploy* de processos de negócios escritos em BPEL, uma vez que dispensa a configuração exaustiva que está associada com outros motores BPEL, como o Apache ODE, em que é necessária a instalação de *servlets* e de um designer, assim como o entrosamento correcto entre estes, o que nem sempre é

fácil, devido a uma série de questões como incompatibilidades de versões, falta de suporte [44], etc.

Outra razão consiste na sua força em termos industriais. Isto porque o ActiveBPEL é a versão freeware de um outro produto da mesma companhia, o ActiveVOS. Isto não quer dizer que o ActiveBPEL seja uma versão deficiente do produto pago, pelo contrário. A versão ActiveVOS apenas possui algumas características e funcionalidades adicionais de mais alto nível, viradas para grandes projectos (como por exemplo mecanismos mais eficazes e completos de controlo de falhas, essencial para projectos empresariais), assim como um suporte mais eficaz e personalizado da ferramenta [43].

Outra das grandes vantagens, já referida anteriormente, é o facto de o motor BPEL ser *open source*, e o designer em si ser *freeware*, permitindo o desenvolvimento de processos de negócio com uma ferramenta completa, livre de custos e funcional. Em relação ao suporte desta versão, consiste num fórum e numa *mailing list*, em que dúvidas dos utilizadores são resolvidas.

Comparado com o NetBeans, o ActiveBPEL fica aquém em termos de usabilidade na criação de processos de negócio, ganhando no entanto no mecanismo de *deploy* destes, que é mais fácil e intuitivo [45].

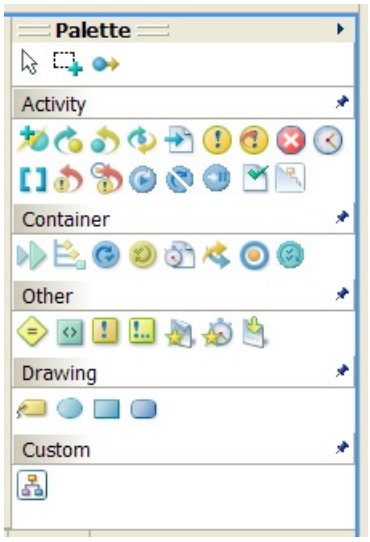
Em relação ao Apache ODE, o ActiveBPEL perde em rapidez (excepto na sua versão empresarial), mas ganha em usabilidade e integração. A outra grande diferente reside no facto de o ActiveBPEL ser um produto disponível sob uma licença *open source*, enquanto que o Apache ODE é ele mesmo um projecto *open source*, logo, com um potencial de desenvolvimento maior, algo que é inerente a este tipo de projectos.

Assim, analisadas as vantagens e desvantagens dos motores, chega-se à conclusão que, para o propósito de uma introdução ao BPEL e à elaboração de processos de negócio, a ferramenta ideal é o ActiveBPEL. Apesar de ficar aquém em certos aspectos em relação aos restantes, é neste momento a ferramenta mais estável, de mais fácil integração e facilidade de desenvolvimento para processos de negócio, sendo uma das propostas de orquestração em BPEL mais neutras e não intrusivas disponíveis.

Capítulo 5 – O ActiveBPEL

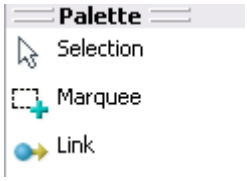
Neste capítulo pretende-se mostrar as actividades que fazem parte da linguagem BPEL, assim como a sua representação no editor ActiveBPEL e utilização. Estas actividades constituem a base do BPEL, sendo necessária a familiarização com estas para se poder esquematizar os processos de negócios que queremos representar.

As funções constituintes da linguagem BPEL são designadas por actividades. Esta ferramenta providencia as funções e componentes necessários à modelação de qualquer processo que se queira realizar. Na palete do ActiveBPEL estão representadas as actividades que constituem a sintaxe do BPEL.



11 – Nas *palettes* presentes no editor encontram-se os objectos que podem ser utilizados para criar processos de negócio.

Na primeira parte da palete, temos o separador "*Links and Selections*", que, apesar de não ser constituído por actividades BPEL, permite escolher as ferramentas de selecção de actividades, assim como a ferramenta utilizada para criar *links* entre actividades.



12 – As ferramentas de selecçao e criação de *links*

De seguida, na *palette*, no separador "*Activity*", encontram-se as principais actividades, as actividades básicas, de um processo BPEL, inclusive as usadas para criar *Fault Handlers*.

Ícone	Instrução	Acção
Invoke	`<invoke>`	Chama um serviço externo de forma síncrona (espera pelo resultado do serviço).
Empty	`<empty>`	É uma actividade "sem operação", útil para sincronizar actividades concorrentes.
Assign	`<assign>`	Manipular variáveis de dados e referências de destino de *partner links*, criando operações de cópia (*copy to/from*) para estes.
Suspend	`<suspend>`	Suspende um processo em execução. Útil numa actividade de *catch* ou *catchAll* para "apanhar" erros inesperados. É uma extensão do BPEL.
Receive	`<receive>`	Recebe entradas de serviços assíncronos.
Reply	`<reply>`	Envia respostas de serviços assíncronos
Wait	`<wait>` `<until>..`	Pausa o fluxo do processo de negócio durante um determinado período de tempo, normalmente para esperar por respostas de outros WS.

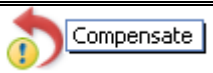
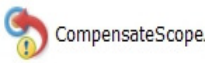
	<exit>	Pára um processo de negócio. As actividades ainda a correr são terminadas.
	<validate>	Valida os valores de variáveis, analisando os seus dados na concordância com a sua definição.
	<continue>	Termina a iteração do ciclo (*while*, *forEach*, *repeatUntil*) em que foi usada e passa para a próxima. É uma extensão do BPEL.
	<break>	Faz com que o ciclo actual (*if*, *while*, *forEach*) termine.
	<throw>	Um meio de lidar com falhas durante o a execução do processo, sinalizando as que ocorrem no seu *scope*.
	<rethrow>	Passa para o *scope* em que está inserida as falhas que são “apanhadas” pelo *fault handler* local.
	<compensate>	Compensa todos os processos internos de um *scope* que já terminaram, quando num deles ocorre um erro, revertendo essas actividades.
	<compensateScope>	Faz a compensação dos processos internos de um *scope* alvo
	<opaque>	Apenas utilizada em processos abstractos (não executáveis), substituindo nestes uma actividade que seria utilizada num processo executável.

Tabela 2 - As actividades básicas usadas em BPEL.

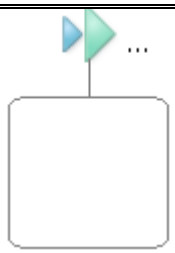
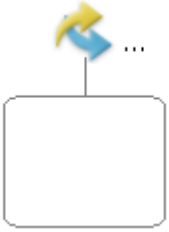
Destas actividades descritas, algumas apenas podem ser usadas em certos contextos num processo. Assim, as actividades de *compensate* e *compensateScope* apenas podem ser usadas dentro de *compensation handlers*

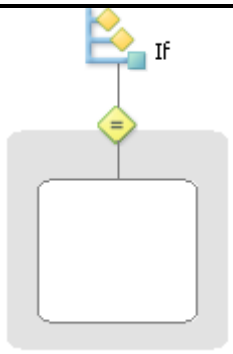
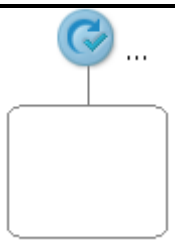
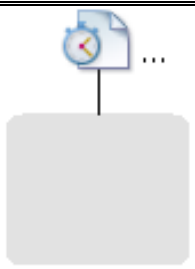
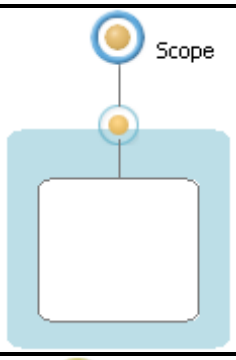
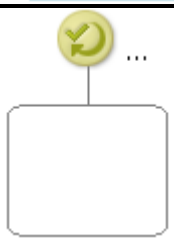
(que têm um *scope* – domínio - onde estão presentes as actividades que têm que compensar), *fault handlers* (responsáveis pelas acções a tomar em caso de falha) ou *termination handlers* (responsável pelas acções a tomar quando um processo ou parte dele é terminado antes que a sua execução esteja completa).

A actividade de *rethrow* apenas pode ser usada dentro de uma actividade de *catch* e *catchAll*, sendo uma das partes standard destas últimas (ambas as actividades são elementos pertencentes aos *fault handlers*).

Quer a actividade *continue* quer *break* estão confinadas a ciclos (*if, while* ou *forEach*), uma vez que a sua acção destina-se a alterar a normal execução destes, terminando uma iteração do ciclo ou o ciclo completo, respectivamente.

No terceiro separador, *container*, estão as actividades que têm associados *scopes* à sua estrutura (denominadas de estruturais), isto é, ciclos, fluxos, sequências, etc, que servem de suporte para um conjunto de actividades dentro dos seus "domínios".

Ícone	Instrução	Acção
	`<sequence>`	Permite definir uma série de actividades que serão invocadas numa sequência ordenada.
	`<flow>`	Executa todas as actividades dentro do seu *scope* em paralelo.

	`<if>`	Executa uma actividade baseando-se numa ou mais condições definidas no elemento "*if*" e nos elementos adicionais "*else if*" e "else". As condições são analisadas por ordem, sendo que a primeira que se verificar terá a actividade correspondente executada
	`<while>`	Executa uma actividade repetidamente, enquanto a condição que lhe está associada for verdadeira.
	`<pick>`	Contém pelo menos uma mensagem ou parte de mensagem e um ou mais alarmes. Quando é executada, esta actividade bloqueia o processo até que a mensagem seja recebida ou, caso contrário, um dos alarmes execute.
	`<scope>`	Providencia um contexto para um conjunto de actividades. Serve para delimitar uma unidade lógica de trabalho, fazendo-a mais acessível de gerir e, caso seja necessário, reverter uma actividade. É um mecanismo de modular actividades.
	`<repeatUntil>`	Executa uma actividade repetidamente até que a sua condição seja verdadeira. Ao contrário da actividade "*while*", a actividade do ciclo é executada pelo menos uma vez.

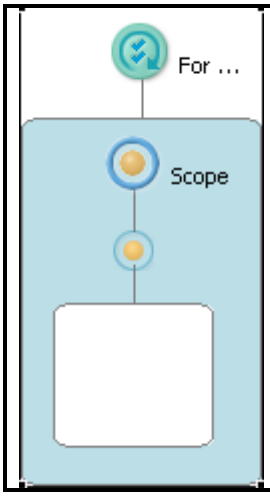
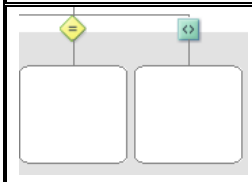
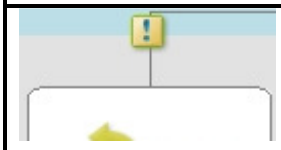
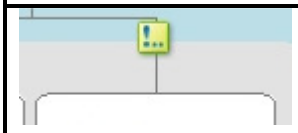
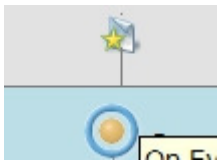
	<forEach>	Funciona na prática como um ciclo "*for*" presente em linguagens como C. As actividades contidas no seu *scope* são executadas um determinado número de vezes, em sequência ou em paralelo. O número de vezes é dado por uma expressão, que vai correr N iterações, em que N varia de um valor inicial a um valor final (normalmente de 1 a N-1).

Tabela 3 - As actividades estruturais usadas em BPEL.

Depois das actividades estruturais, aparece na *palette* um separador (*other*) com actividades complementares.

Ícone	Instrução	Acção
	<elseif> <else>	Definem condições extra num ciclo *if*. Podem existir vários *elseIf* por ciclo, e um *else*, normalmente usado como a escolha por defeito.
	<catch>	Um *fault handler*, que define no seu scope as actividades a executar caso uma falha com um nome e/ou tipo especifico ocorra.
	<catchAll>	Um *fault handler* que executa quando uma actividade de *throw* não é apanhada por nenhum *catch*. Só pode existir um por *fault handler.*
	<onEvent>	Indica que o evento especificado no seu *scope* está à espera de uma mensagem para ser despoletado, quer esta seja de uma actividade *receive/reply* ou operação de sentido único (*one-way*). Ao contrário de uma actividade de *receive*, não é

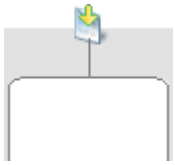
		criada uma instância do processo.
	<onAlarm>	Define um evento temporal de *timeout*, quer através de uma duração, quer através de um limite (*deadline*), despoletando um alarme quando o *timeout* ocorre.
	<onMessage>	Define um conjunto de actividades a executar aquando da recepção de uma mensagem. É uma particularização de uma actividade *onEvent*.

Tabela 4 - Elementos do quarto separador da *palette* (*Others*).

Os elementos deste separador são assim elementos de suporte para outras actividades, permitindo aumentar a complexidade e profundidade dos processos de negócio desenvolvidos pelo utilizador.

Com os elementos presentes nestas "*palettes*" e com WS externos podem-se criar processos de negócio, dos mais simples aos mais complexos, invocando, gerindo, isto é, orquestrando vários WS para que sejam atingidos os fins a que um projecto BPEL se propõe – a criação de um processo de negócio – também ele disponível sob a forma de WS.

Capítulo 6 – Exemplos de processos de negócio

Qual é a aparência de um processo de negócio? Qual é o código XML que lhe serve de base? Como são construídos os elementos, as invocações e as ligações entre estes?

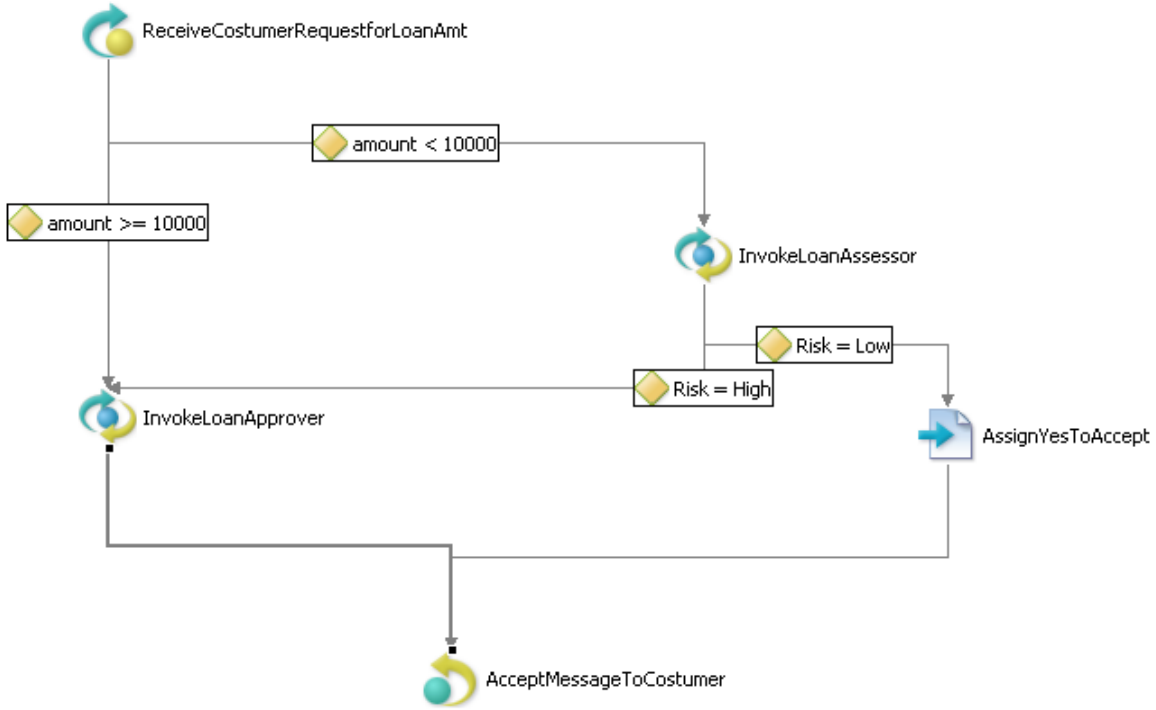
Neste capítulo irão ser mostrados exemplos de processos de negócio, analisando-os em termos de BPEL, usando representações criadas no ActiveBPEL *Designer* e explicando o código que está na sua base.

Os processos de negócio analisados de seguida mostram o uso da maior parte das actividades de BPEL, de modo a exemplificar a sua importância dentro de um processo, a sua utilidade e a sua utilização.

O primeiro processo foi realizado tendo por base um tutorial da aplicação activeBPEL, de modo a servir como iniciação à ferramenta e suas capacidades, assim como na sintaxe do BPEL.

Os dois processos de negócio seguintes foram retirados de um conjunto disponibilizado no site da activeEndpoints [46], a companhia responsável pelo desenvolvimento e suporte do activeBPEL, e representam uma série de actividades e mecanismos que podem ser usados em processos mais complexos, apresentando uma série de funcionalidades úteis a estes.

6.1 – Aprovação de um empréstimo.

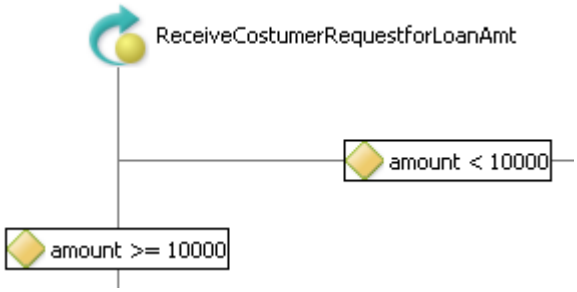


13 – Processo de negócio de um pedido de empréstimo.

Este é um processo de negócio simples, que consiste numa aprovação de empréstimo, tendo como critérios de aprovação a quantia pretendida e o risco que um dado cliente representa.

Este processo é iniciado com a recepção de um pedido do cliente com o valor do empréstimo pretendido. Se esse valor é inferior a 10000, é invocado um WS que representa o assessor do empréstimo, e que vai verificar se o cliente é de alto ou baixo risco. Se o risco é baixo, o empréstimo é feito com sucesso, caso contrário é enviado para um novo WS, o *LoanApprover*, que vai aprovar ou não o empréstimo. Caso o valor do pedido seja superior ou igual a 10000, o pedido é imediatamente enviado para o *LoanApprover*. O cliente receberá uma aceitação por parte do assessor ou um sim/não por parte do *Approver*.

O processo começa assim com uma instrução de *receive*, que vai escolher o caminho tomado pelo pedido baseando-se no montante:



14 – A primeira escolha a ser feita tem por base a quantia pretendida para o empréstimo.

```
<bpel:receive createInstance="yes"
name="ReceiveCostumerRequestforLoanAmt" operation="request"
partnerLink="costumer" portType="lns:loanServicePT" variable="request">
         <bpel:sources>
            <bpel:source linkName="receive-to-assess">
               <bpel:transitionCondition>$request.amount &lt;
10000</bpel:transitionCondition>
            </bpel:source>
            <bpel:source linkName="receive-to-aprove">
               <bpel:transitionCondition>$request.amount &gt;=
10000</bpel:transitionCondition>
            </bpel:source>
         </bpel:sources>
      </bpel:receive>
```

No código pode ler-se a instrução inicial *receive*, delimitada pelas expressões "`<bpel:receive`" e "`</bpel:receive>`", seguida dos argumentos necessários à sua criação, como o seu nome ("`ReceiveCostumerRequestforLoanAmt`"), o nome da operação (`"request"`), do *Partner Link* a que esta associada ("`costumer`"), o *portType* utilizado ("`lns:loanServicePT`") e o nome da variável (que vai ter neste caso o mesmo nome que a operação - `"request"`).

Dentro do corpo da instrução, situam-se as condições que vão definir o caminho a seguir pelo pedido.

```
<bpel:transitionCondition>$request.amount &lt;
10000</bpel:transitionCondition>
            </bpel:source>
```

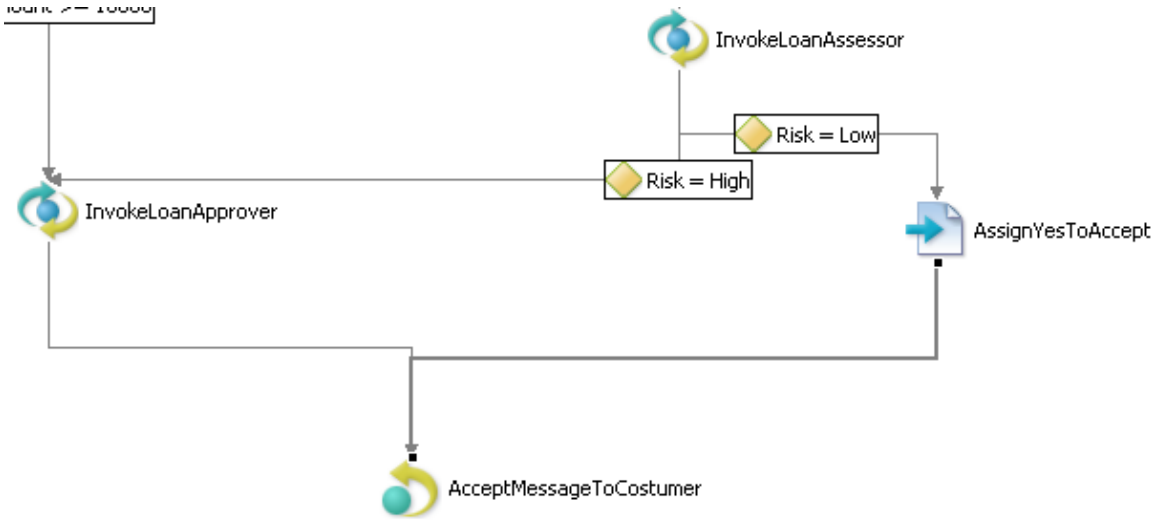
```
            <bpel:source linkName="receive-to-aprove">
               <bpel:transitionCondition>$request.amount &gt;=
10000</bpel:transitionCondition>
```

Estas condições são demarcadas pela primitiva BPEL "`bpel:transitionCondition`", em que o campo "*amount"* do pedido é comparado com um valor (10000), sendo que o resultado dessa comparação leva à escolha do caminho deste.

Como se pode ver pelo código, todas as primitivas e construções BPEL estão claramente assinaladas com a expressão "`<bpel:…`", tornando-as assim distintas de código XML normal.

De seguida, se o pedido for inferior a 10000, é invocado um WS, com o nome "InvokeLoanAssessor", que vai fazer a avaliação do pedido dependendo do risco do cliente.

Caso o risco seja "*Low*", é feito um *assingn* com a resposta positiva para o cliente, caso contrário é necessário invocar um novo WS, *InvokeLoanApprover*, que também é invocado quando o valor do empréstimo pretendido excede ou iguala o valor de 10000, que irá ser responsável pela decisão final em relação à



15 - O Web Service invocado, *invokeLoanAssessor*, decide o destino do pedido, de acordo com o risco associado ao cliente.

aceitação ou não do pedido de empréstimo.

```
<bpel:invoke inputVariable="request" name="InvokeLoanAssessor"
operation="check" outputVariable="risk" partnerLink="assessor"
portType="lns:riskAssessmentPT">
         <bpel:targets>
            <bpel:target linkName="receive-to-assess"/>
         </bpel:targets>
         <bpel:sources>
            <bpel:source linkName="assess-to-aprove">
               <bpel:transitionCondition>$risk.level !=
'low'</bpel:transitionCondition>
            </bpel:source>
            <bpel:source linkName="assess-to-setMessage">
               <bpel:transitionCondition>$risk.level =
'low'</bpel:transitionCondition>
            </bpel:source>
         </bpel:sources>
</bpel:invoke>
```

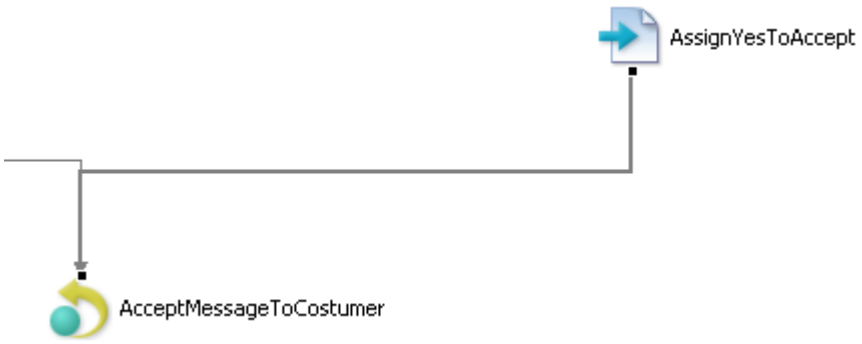
Neste caso temos uma instrução de *invoke*, delimitada pelas primitivas "`<bpel:invoke…`" e "`</bpel:invoke>`"; no cabeçalho da instrução, estão presentes as variáveis de entrada, o nome do WS (`"InvokeLoanAssessor"`), assim como a operação, a variável de saída, o *Partner Link* e o tipo do porto utilizado. De notar que os campos presentes no cabeçalho não diferem dos relativos a instruções de *receive*/*reply*.

```
<bpel:targets>
            <bpel:target linkName="receive-to-aprove"/>
            <bpel:target linkName="assess-to-aprove"/>
</bpel:targets>
```

Os *links* de entrada são designados por *targets*, aparecendo logo a seguir ao cabeçalho da actividade, sendo todos definidos dentro do mesmo contexto (limitados pelas instruções `<bpel:targets>` e `</bpel:targets>`).

```
<bpel:sources>
            <bpel:source linkName="assess-to-aprove">
               <bpel:transitionCondition>$risk.level !=
'low'</bpel:transitionCondition>
            </bpel:source>
            <bpel:source linkName="assess-to-setMessage">
               <bpel:transitionCondition>$risk.level =
'low'</bpel:transitionCondition>
            </bpel:source>
</bpel:sources>
```

As *sources* presentes no código representam os *links* de saída da instrução (são delimitados pelas instruções `<bpel:sources>` e `</bpel:sources>`. Para cada *link* temos associadas as transições/condições que representam, dentro de um contexto próprio para cada *source*. Neste caso, tem-se que, se o risco é "Low" (`$risk.level = 'low'`) o processo continua pelo *link* `"assess-to-setMessage"`, que irá fazer um *assign* da resposta para ser transmitida à saída do processo; caso contrário (`$risk.level != 'low'`) o *link* seguido será `"assess-to-aprove"`, que irá encaminhar o pedido para o serviço "*InvokeLoanApprover*".



16 – Parte final do processo. A resposta é copiada para a variável de saída através de um *assign*, sendo enviada para o utilizador através de um *reply* ("*AcceptMessageToCostumer*")

```
<bpel:assign name="AssignYesToAccept">
         <bpel:targets>
            <bpel:target linkName="assess-to-setMessage"/>
         </bpel:targets>
         <bpel:sources>
            <bpel:source linkName="L2"/>
         </bpel:sources>
         <bpel:copy>
            <bpel:from>
               <bpel:literal>Yes</bpel:literal>
            </bpel:from>
            <bpel:to part="accept" variable="approval"/>
         </bpel:copy>
</bpel:assign>
```

O *assign* apenas possui no seu cabeçalho o seu nome (`"AssignYesToAccept"`). Novamente se encontram definidos os *links* de entrada e saída, *target* e *source* respectivamente, seguidos da actividade a que este se destina, a cópia de um valor para a variável de saída.

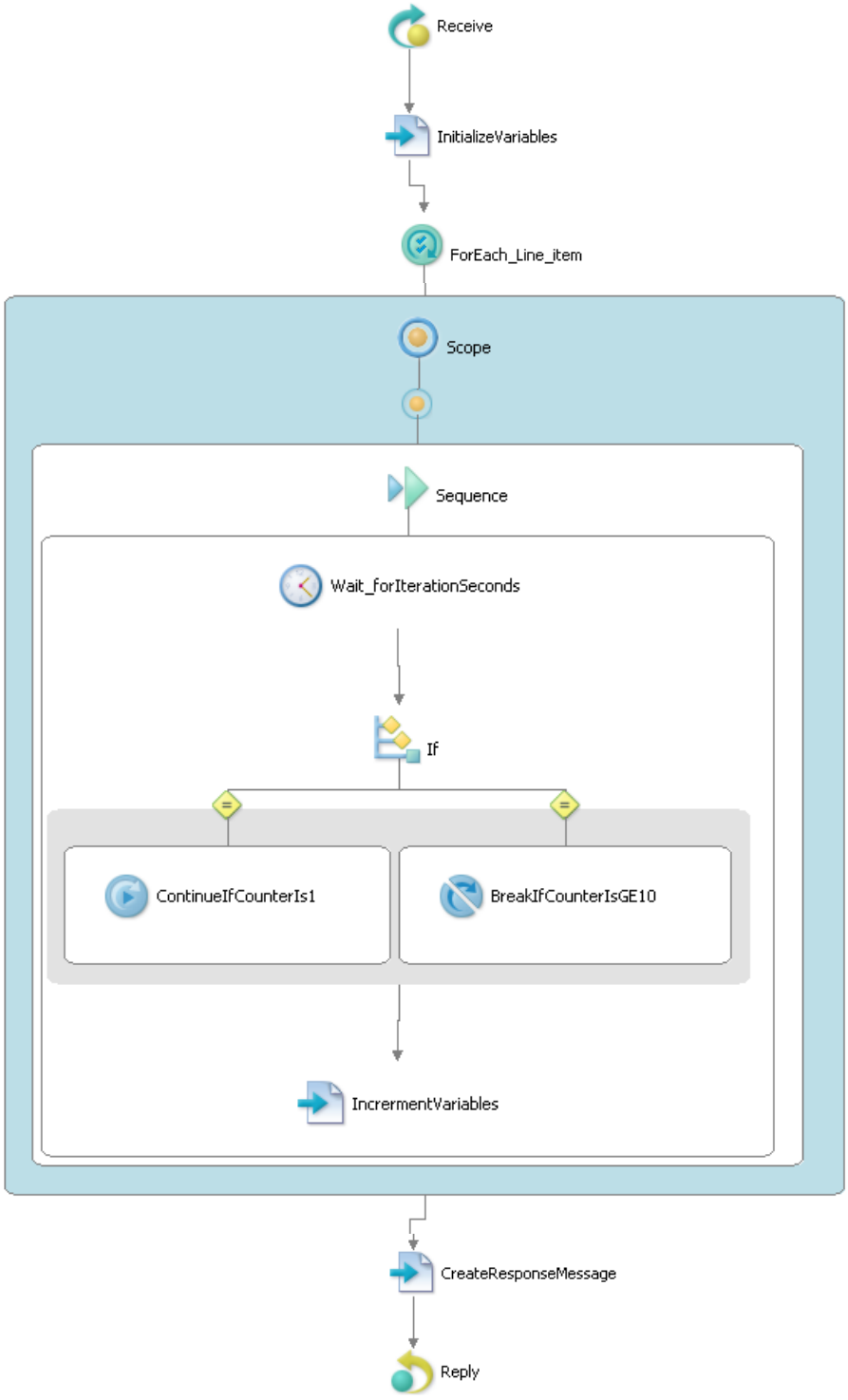
```
<bpel:copy>
            <bpel:from>
               <bpel:literal>Yes</bpel:literal>
            </bpel:from>
            <bpel:to part="accept" variable="approval"/>
</bpel:copy>
```

A cópia (delimitadas pelas instruções `<bpel:copy>` e `</bpel:copy>`) é feita definindo a origem do valor (`<bpel:from>`), neste caso o literal "Yes", e a variável de destino, assim como o campo da variável em específico (`<bpel:to part="accept" variable="approval"/>`).

```
<bpel:reply name="AcceptMessageToCostumer" operation="request"
partnerLink="costumer" portType="lns:loanServicePT" variable="approval">
         <bpel:targets>
            <bpel:target linkName="L1"/>
            <bpel:target linkName="L2"/>
         </bpel:targets>
</bpel:reply>
```

Para finalizar o processo, tem-se uma actividade de *reply*, que vai devolver os valores de resposta do processo. Possui o cabeçalho com os mesmos campos que as actividades de *request* e *invoke* e, uma vez que não possui *links* de saída (é a actividade final do processo), apenas tem os *targets* que correspondem aos links de entrada.

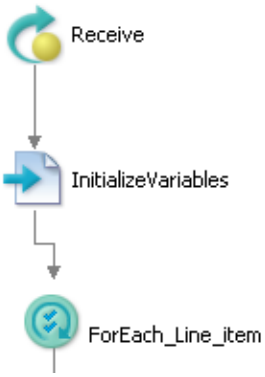
6.2 – Ordem de encomenda



17 – Mapa do processo de negócio de processamento de encomendas.

Este processo destina-se a mostrar o funcionamento de vários tipos de actividades presentes na linguagem BPEL, algumas delas, como o *forEach*, *continue* e *break* foram implementadas na versão 2.0 do BPEL, que neste momento é o standard da linguagem. O processo consiste em percorrer uma série de elementos de uma ordem de encomenda, contando-os à medida que estes são adereçados e atendidos, e dependendo desse número, continuar a percorrer elementos ou termina-o.

O processo começa com um *receive*, que aceita a ordem de encomenda, sendo a resposta devolvida por um *reply* no final do processo, tal como na actividade mostrada anteriormente.



18 – O processo é iniciado com a recepção da ordem. Em seguida são inicializadas as variáveis que irão ser utilizadas, seguindo-se a actividade *forEach*.

De seguida são inicializadas as variáveis que vão ser utilizadas no processo utilizando uma instrução de *assign* denominada por "*InitializeVariables*". Este processo é feito através da cópia de valores para as variáveis.

```
<bpel:assign name="InitializeVariables">
         <bpel:targets>
            <bpel:target linkName="L1"/>
         </bpel:targets>
         <bpel:sources>
            <bpel:source linkName="L3"/>
         </bpel:sources>
         <bpel:copy>
            <bpel:from>count($orderMessage.order/OrderDetail
)</bpel:from>
            <bpel:to variable="numberOfDetails"/>
         </bpel:copy>
         <bpel:copy>
```

```
      <bpel:from>0</bpel:from>
      <bpel:to variable="numberOfCountedDetails"/>
   </bpel:copy>
   <bpel:copy>
      <bpel:from>0</bpel:from>
      <bpel:to variable="totalOrderValue"/>
   </bpel:copy>
      </bpel:assign>
```

São três as variáveis inicializadas: `"numberOfDetails"`, que representa o número de elementos iniciais da ordem de encomenda (o número inicial é obtido pela expressão `count($orderMessage.order/OrderDetail)`; `"numberOfCountedDetails"`, que vai contar o número de detalhes/elementos da ordem que foram processados (inicializado a 0); `"totalOrderValue"`, que vai guardar o valor total dos elementos da encomenda processados (que também é inicializado a 0).

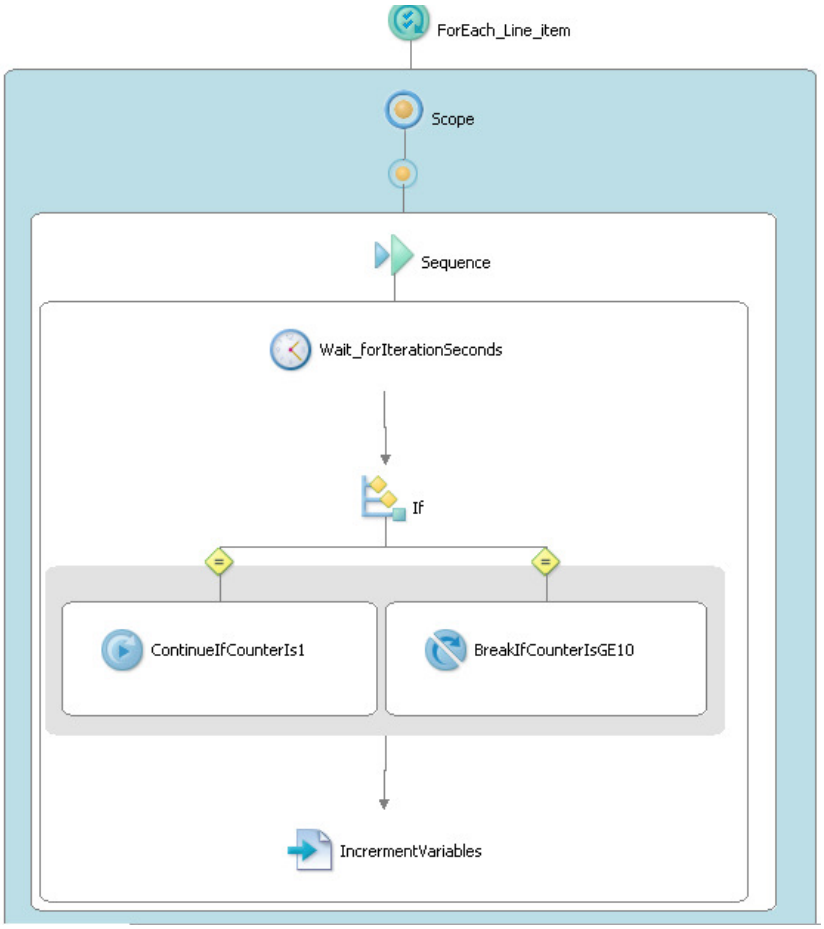
Depois das variáveis inicializadas, segue-se a instrução que dá o nome ao processo, *forEach*, que vai percorrer cada elemento da ordem da encomenda e aplicar em cada um as instruções que estão no corpo da instrução.

```
      <bpel:forEach counterName="counter" name="ForEach_Line_item"
parallel="yes">
         <bpel:targets>
            <bpel:target linkName="L3"/>
         </bpel:targets>
         <bpel:sources>
            <bpel:source linkName="L2"/>
         </bpel:sources>
         <bpel:startCounterValue>1</bpel:startCounterValue>
        <bpel:finalCounterValue>$numberOfDetails</bpel:finalCounterValue>
```

O cabeçalho desta actividade contém o seu nome (`name="ForEach_Line_item"`), o nome do contador utilizado (`counterName="counter"`) e a escolha da opção se cada "instância" irá ser corrida em paralelo com as restantes, que neste caso é afirmativo (`parallel="yes"`). As entradas para a instrução, *targets*, e as saídas, *sources*, encontram-se definidas logo de seguida. Também o contador fica definido com o número inicial (`<bpel:startCounterValue>1</bpel:startCounterValue>`) e final (que

corresponde ao número de detalhes/elementos da ordem, `<bpel:finalCounterValue>$numberOfDetails</bpel:finalCounterValue>`).

Todo o corpo da instrução se encontra dentro do seu "*scope*", isto é, o conjunto de instruções a ser aplicado a cada elemento funciona como uma "caixa negra" independente, em que cada um é tratado independentemente, percorrendo todo o fluxo de instruções até ser devolvida uma resposta.



19 – No corpo, *scope*, da instrução *forEach,* está definida a sequência pela qual vao passar os elementos da ordem de encomenda

Neste caso, o corpo da instrução consiste numa sequência (definida pelo cabeçalho `<bpel:sequence>`).

```
<bpel:wait name="Wait_forIterationSeconds">
      <bpel:for>concat('PT', string($counter), 'S')</bpel:for>
</bpel:wait>
```

Esta começa com uma actividade de *wait*, que usa uma expressão baseada no contador, para definir qual o tempo a esperar, de modo a simular o efeito de um *invoke* a ocorrer em cada uma das iterações da actividade do *forEach*.

```
<bpel:if>
      <bpel:condition>($counter = 1)</bpel:condition>
      <bpel:extensionActivity>
           <ext1:continue name="ContinueIfCounterIs1"/>
      </bpel:extensionActivity>
      <bpel:elseif>
           <bpel:condition>($counter &gt;= 10)</bpel:condition>
           <bpel:extensionActivity>
                <ext1:break name="BreakIfCounterIsGE10"/>
           </bpel:extensionActivity>
       </bpel:elseif>
</bpel:if>
```

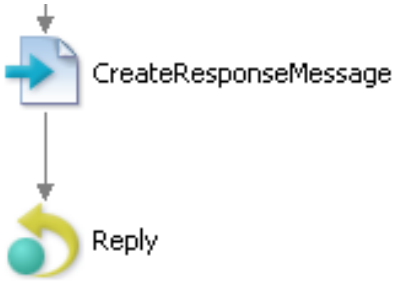
Segue-se uma instrução condicional, um *if*, que vai analisar o valor do contador (`$counter`). Se o contador é igual a 1 (`<bpel:condition>($counter = 1)</bpel:condition>`) é executada uma actividade de *continue* (`"ContinueIfCounterIs1"`), que termina todas as instruções ainda a correr da iteração, completa o *scope* normalmente, e continua com a próxima iteração (execução sequencial) ou permite que todas as outras iterações continuem (execução paralela, como é o caso neste processo).

Quando o contador chega a 10 (`<bpel:condition>($counter >= 10)</bpel:condition>`), é executada uma actividade de *break* (`"BreakIfCounterIsGE10"`), que, tal como a instrução de *continue*, termina as instruções ainda a correr na iteração e completa o *scope* normalmente, mas que completa a actividade *forEach* sem terminar quaisquer instruções ainda em espera (execução sequencial) ou, neste caso (execução paralela), força o término de todas as instruções ainda a correr.

```
<bpel:assign name="IncrermentVariables">
                  <bpel:copy>
                     <bpel:from>($numberOfCountedDetails + 1)</bpel:from>
                     <bpel:to variable="numberOfCountedDetails"/>
                  </bpel:copy>
                  <bpel:copy>
```

```
<bpel:from>($orderMessage.order/OrderDetail[$counter]/TotalCost +
$totalOrderValue)</bpel:from>
                    <bpel:to variable="totalOrderValue"/>
                </bpel:copy>
</bpel:assign>
```

A terminar a sequência, aparece uma actividade de *assign* (`"IncrermentVariables"`), que é responsável por incrementar o número de elementos processados (`"numberOfCountedDetails"`), assim como de actualizar o valor total da encomenda (`"totalOrderValue"`), à medida que os elementos desta vão sendo processados, incrementando o valor de cada um destes ao total (`($orderMessage.order/OrderDetail[$counter]/TotalCost +$totalOrderValue)`).



20 –Para finalizar o processo, um *assign* cria a resposta para o cliente, que é de seguida devolvida através de um *reply*.

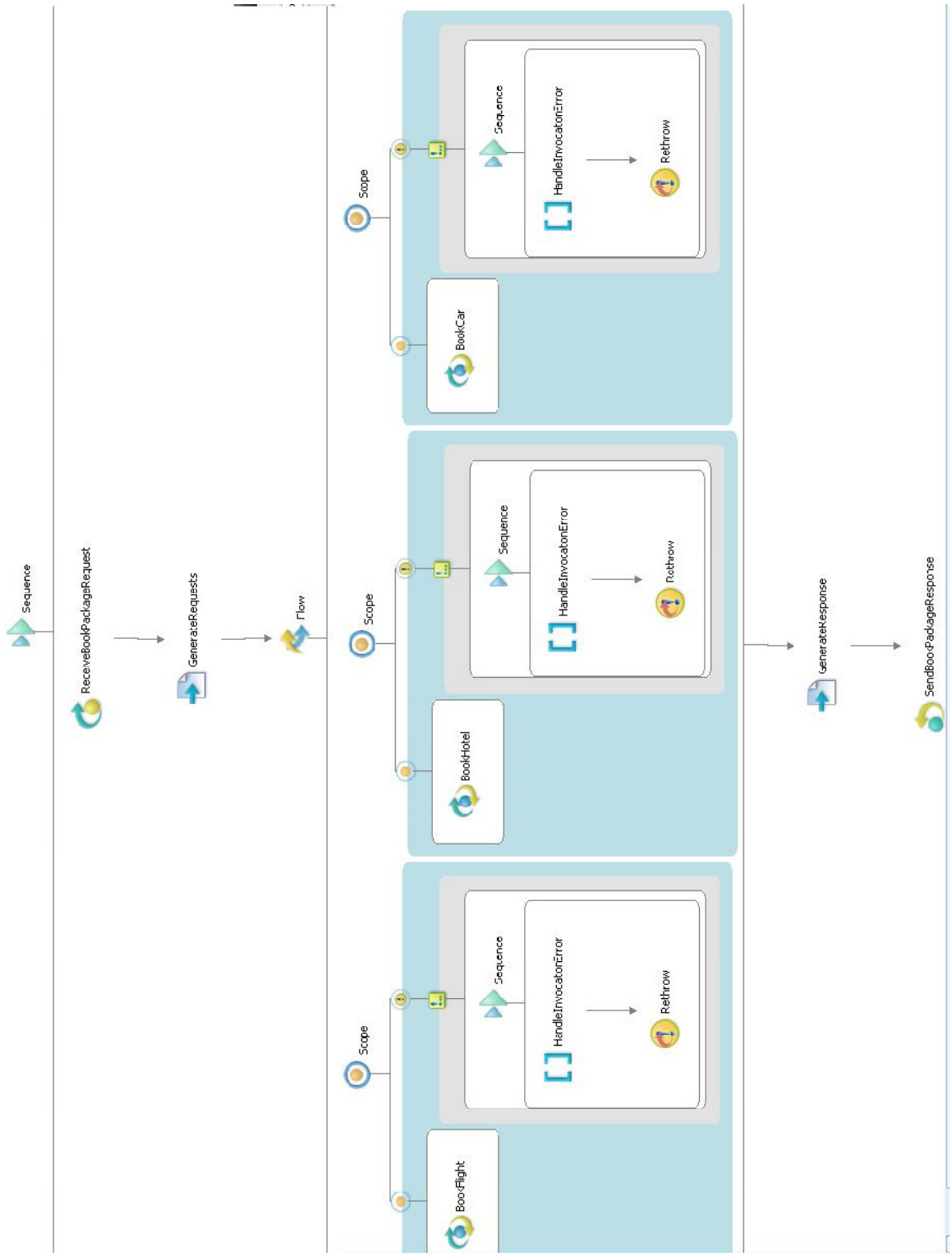
```
<bpel:assign name="CreateResponseMessage">
         <bpel:targets>
            <bpel:target linkName="L2"/>
         </bpel:targets>
         <bpel:sources>
            <bpel:source linkName="L4"/>
         </bpel:sources>
         <bpel:copy>
            <bpel:from>concat('The order contains ', $numberOfDetails, '
detail elements and the proceess traversed ', $numberOfCountedDetails, '
detail elements.  The total calculated order value =  $',
$totalOrderValue)</bpel:from>
            <bpel:to part="response" variable="orderProcessResponse"/>
         </bpel:copy>
         <bpel:copy>
            <bpel:from part="order" variable="orderMessage">
               <bpel:query>OrderHeader/PONo</bpel:query>
            </bpel:from>
            <bpel:to part="PONum" variable="orderProcessResponse"/>
         </bpel:copy>
         <bpel:copy>
            <bpel:from part="order" variable="orderMessage">
               <bpel:query>OrderHeader/CustId</bpel:query>
            </bpel:from>
            <bpel:to part="CustID" variable="orderProcessResponse"/>
```

```
        </bpel:copy>
</bpel:assign>
```

No fim do processo, depois da actividade *forEach* estar concluída, existe um *assign* (`CreateResponseMessage`) que vai construir a mensagem que irá ser devolvida ao cliente, que terá a forma "*The order contains N detail elements and the proceess traversed M detail elements. The total calculated order value = $Y*", que é obtido usando uma instrução de concatenação, `concat(...)`, para juntar as partes da mensagem. Para além da mensagem, todas as outras variáveis de retorno são preenchidas com os valores calculados (`part="PONum" variable="orderProcessResponse";` `part="CustID" variable="orderProcessResponse"`), usando em todos os casos instruções de cópia.

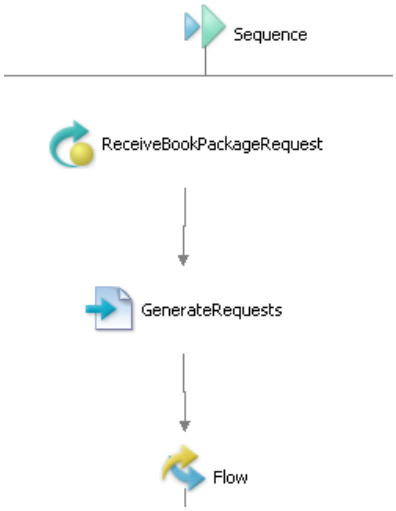
As variáveis são por fim retornadas ao cliente através da actividade de *reply*, a parte final da actividade *receive/reply* que serviu de base ao processo de negócio.

6.3 – Marcação de viagem



21 - Esquema do processo de negócio que permite marcar uma viagem, reservando o voo, o hotel e o carro.

Este processo de negócio permite a reserva de três elementos de uma viagem: os bilhetes de avião, o quarto de hotel e o carro de aluguer, devolvendo ao cliente o resultado das reservas. O processo invoca vários serviços que vão, cada um, tratar de uma das partes da reserva, possuindo também mecanismos de tratamento de falhas, ou *fault handlers*. Todo o processo está inserido numa actividade de *sequence*, o que quer dizer que todas as suas actividades são executadas de forma sequencial, isto é, apenas quando a actividade anterior terminar é que a seguinte é iniciada. O cabeçalho da sequência, em XML, apenas possui a indicação de inicio de actividade, sem nome (apesar deste poder ser definido) nem parâmetros de entrada ou saída.

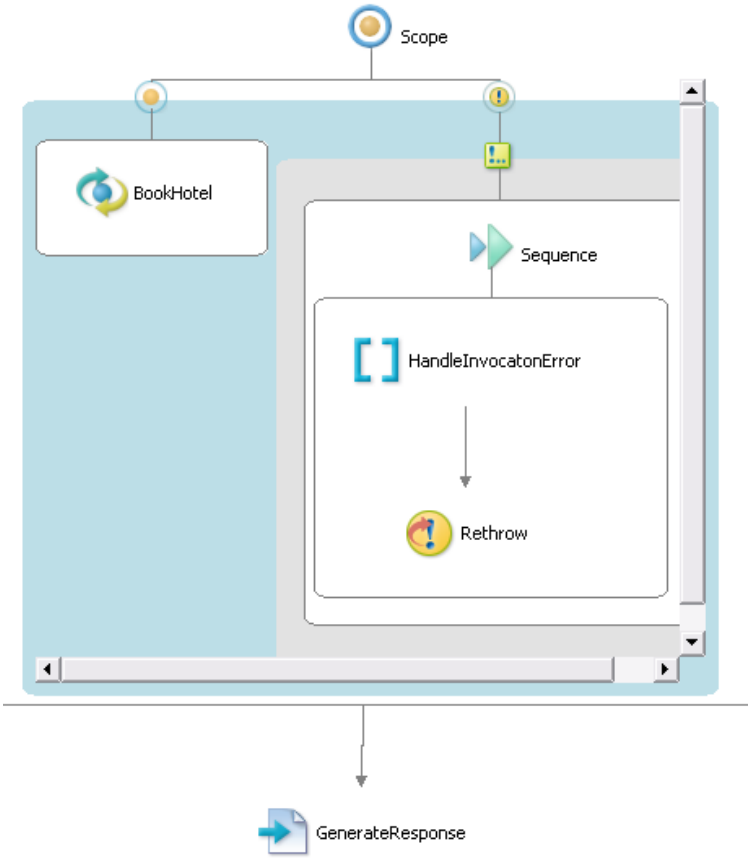


22 – Todo o processo se encontra dentro de uma actividade de *sequence*. A execução é iniciada com um *receive* seguido de um *assign* que vai copiar os valores do pedido para os vários Web Services que irão ser invocados.

```
<bpel:sequence>
      <bpel:receive createInstance="yes" name="ReceiveBookPackageRequest"
operation="BookPackage" partnerLink="BookPackageLink"
portType="ts:TravelServicePT" variable="BookPackageRequest"/>
      <bpel:assign name="GenerateRequests">
         <bpel:copy>
            <bpel:from part="Document" variable="BookPackageRequest"/>
            <bpel:to part="Document" variable="BookFlightRequest"/>
         </bpel:copy>
         <bpel:copy>
            <bpel:from part="Document" variable="BookPackageRequest"/>
```

```
      <bpel:to part="Document" variable="BookHotelRequest"/>
    </bpel:copy>
    <bpel:copy>
      <bpel:from part="Document" variable="BookPackageRequest"/>
      <bpel:to part="Document" variable="BookRentalCarRequest"/>
    </bpel:copy>
  </bpel:assign>
```

O processo é iniciado com a recepção de um pedido de marcação de uma viagem pelo cliente. A recepção é feita por uma actividade de *receive*, "ReceiveBookPackageRequest”,



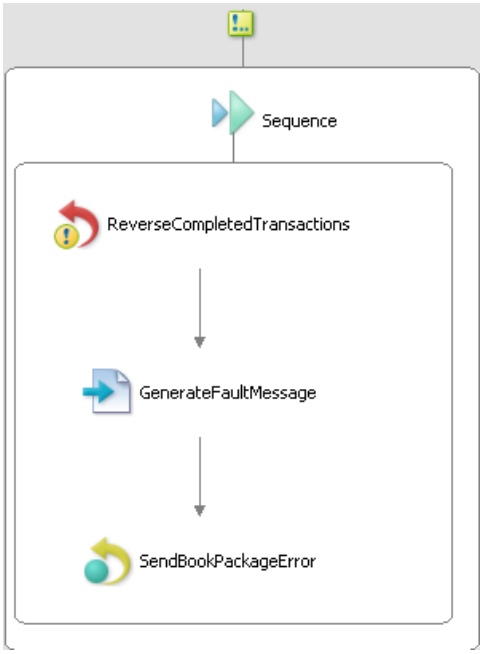
23 – Dentro do fluxo do processo são invocados três *Web Services*. Cada um dentro de um *scope*, que engloba não só a invocação como a lógica necessária para apanhar e lidar com falhas.

Dentro do fluxo podem-se encontrar três *scopes* independentes, cada um deles contendo não só a invocação do WS correspondente, como também toda a

lógica (conjunto de instruções) responsável por encontrar e lidar com eventuais falhas no processo de invocação.

```
<bpel:scope>
            <bpel:faultHandlers>
               <bpel:catchAll>
                  <bpel:sequence>
                     <bpel:empty name="HandleInvocatonError"/>
                     <bpel:rethrow/>
                  </bpel:sequence>
               </bpel:catchAll>
            </bpel:faultHandlers>
                  <bpel:invoke inputVariable="BookHotelRequest"
      name="BookHotel" operation="BookHotel"
      outputVariable="BookHotelResponse" partnerLink="BookHotelLink"
      portType="ts:HotelServicePT"/>
            </bpel:scope>
```

Da análise do código XML pode-se ver a definição dos *Fault Handlers*, que utiliza uma actividade *catchAll* para receber qualquer falha que ocorra durante o processo de invocação do WS. Dentro do *scope* do *catchAll* encontra-se um processo vazio, onde a lógica específica de tratamento da falha, caso esta acontecesse numa aplicação real, seria escrita; seguida de uma actividade de *rethrow*, que vai lançar a falha para o *scope* em que está inserido, de modo a ser também tratada pelo *Fault Handler* do processo.

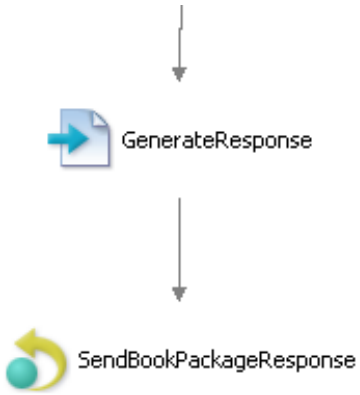


24 – Gestor de falhas (*Fault handler)* do processo.

```
<bpel:faultHandlers>
      <bpel:catchAll>
         <bpel:sequence>
            <bpel:compensate name="ReverseCompletedTransactions"/>
            <bpel:assign name="GenerateFaultMessage">
               <bpel:copy>
                  <bpel:from>
                     <bpel:literal>An error occurred while submitting the
order. All transactions have been canceled.
</bpel:literal>
                  </bpel:from>
                  <bpel:to part="Document"
variable="BookPackageFaultVar"/>
               </bpel:copy>
            </bpel:assign>
            <bpel:reply faultName="ts:BookPackageFault"
name="SendBookPackageError" operation="BookPackage"
partnerLink="BookPackageLink" portType="ts:TravelServicePT"
variable="BookPackageFaultVar"/>
         </bpel:sequence>
      </bpel:catchAll>
         </bpel:faultHandlers>
```

Este *Fault Handler* vai receber todas as falhas ocorridas durante a execução do processo e gerar uma mensagem de erro para o utilizador, ao mesmo tempo que cancela todas as actividades deste mesmo as já concluídas, de modo a que, se, por exemplo, a reserva de um carro falhar e as reservas de hotel e bilhetes de avião já tiverem sido concluídas, estas sejam desfeitas, ficando sem validade (algo que é feito com uma actividade de *compensate*, `"ReverseCompletedTransactions"`).

Este processo mostra assim muitos dos mecanismos do BPEL existentes para lidar com falhas nos processos, mecanismos estes que podem ser locais, numa determinada actividade (como neste caso nos *invokes* dos WS) ou trecho do processo; como globais, gerindo todas as falhas do processo de um modo geral, usando *Fault Handlers* ligados ao processo por inteiro.



25 – No final do processo é escrita uma mensagem com os dados das reservas para o cliente ou com um *log* dos erros que ocorreram, em caso de falha

```
<bpel:assign name="GenerateResponse">
         <bpel:copy>
            <bpel:from>concat( 'Flight: ', $BookFlightResponse.Document ,
'- Hotel: ', $BookHotelResponse.Document , '- Rental Car: ',
$BookRentalCarResponse.Document )</bpel:from>
            <bpel:to part="Document" variable="BookPackageResponse"/>
         </bpel:copy>
      </bpel:assign>
      <bpel:reply name="SendBookPackageResponse" operation="BookPackage"
partnerLink="BookPackageLink" portType="ts:TravelServicePT"
variable="BookPackageResponse"/>
         </bpel:sequence>
```

Para finalizar, uma actividade de *reply* vai devolver a resposta gerada no *assign*, `"GenerateResponse"`, com os dados das reservas efectuadas (`concat( 'Flight: ', $BookFlightResponse.Document , '- Hotel: ', $BookHotelResponse.Document , '- Rental Car: ', $BookRentalCarResponse.Document )`) ou, caso tenha ocorrido uma falha no processo, um *log* do erro.

Capítulo 7 – Conclusões e trabalho futuro

Ao realizar este documento, conclui-se que o BPEL é uma linguagem com muito potencial no desenvolvimento de processos de negócio e *Web Services* em geral, potencial esse que já começa a ser aproveitado pela indústria, de um modo cada vez mais profundo.

Apesar de relativamente recente (a versão 2.0, a primeira como standard, foi lançada em Agosto de 2007), tem uma grande vantagem em relação às concorrentes – o suporte da indústria informática e das grandes companhias de desenvolvimento de software em particular. O facto de ser baseada em tecnologias estáveis e largamente utilizadas, como SOAP, WSDL e UDDI, aumenta ainda mais a sua estabilidade e usabilidade, existindo já várias ferramentas, motores e ambientes de integração que suportam a sua sintaxe e mesmo dedicados apenas a processos em BPEL. A conhecidos IDE's, como Eclipse e Netbeans, juntam-se cada vez mais ofertas de IDE's desenvolvidos especialmente para BPEL, com suporte para todas as actividades presentes na versão 2.0 da linguagem (um dos melhores exemplos disso é o activeBPEL, mostrado neste documento).

Também o suporte por parte da comunidade de programadores tem vindo a crescer, com cada vez mais fóruns de apoio e páginas online sobre a linguagem, resolução de problemas ou mesmo desenvolvimento dos motores existentes, que são na sua maioria *open source*.

A massificação da internet está a levar a que muitos dos serviços que tradicionalmente seria desenvolvidos e utilizados em cada empresa comecem a aparecer como serviços Web, desenvolvidos e disponibilizados gratuitamente, ou mediante alguma forma de pagamento ou fidelização, para serviços mais complexos, de modo a que qualquer utilizador, particular ou empresarial, possa desenvolver novos serviços ou simplesmente utilizar os já existentes para suprir as suas necessidades. Deste modo, o BPEL surge na vanguarda da utilização

desses serviços, e quem estiver familiarizado com a sua utilização estará sempre um passo à frente da concorrência, rentabilizando os seus recursos e criando soluções cada vez mais rápidas, completas e personalizáveis para as suas necessidades.

Os motores e ferramentas estudados demonstram já uma certa maturidade, no caso do activeBPEL e Netbeans, e bastante potencial, no caso do ODE. Dos três, o activeBPEL é a melhor ferramenta para quem quer iniciar o seu percurso na linguagem BPEL, uma vez que possui um IDE funcional e atractivo, baseado no IDE Java Eclipse, que possui uma grande comunidade de utilizadores, que continuam a fazer grandes contribuições para esta ferramenta; com um pacote de componentes de fácil instalação, já com todos os requerimentos incorporados de modo a permitir uma abordagem "*hands on*" (instalar e começar a trabalhar) ao BPEL. A ferramenta Netbeans também apresenta uma boa implementação do BPEL, com um editor funcional, mas peca pela menor facilidade de desenvolvimento de projectos em relação ao activeBPEL, sendo um pouco mais confusa e com um método de *deploy* dos projectos menos prática e mais susceptível a erros. Finalmente, o Apache ODE, que, apesar de mostrar bastante potencial, sendo já um dos motores mais rápidos de BPEL, ainda não se encontra totalmente desenvolvido, sendo ainda muito susceptível a incompatibilidades; isto devido ao facto de não possuir um IDE próprio, sendo apenas um motor para integração de BPEL. Assim, por depender de IDE´s e *servlets* externos, está sujeito às limitações que estes apresentam, especialmente incompatibilidades entre estes. No entanto, como continua a ser um projecto activo e em pleno crescimento, dentro de pouco tempo irá ser visto e usado como um dos principais motores BPEL.

Os serviços desenvolvidos durante a realização deste documento não são aplicáveis a uma área em específico, sendo no entanto uma introdução para a sua implementação em qualquer área com interacção com a Web.

Uma vez que, dada a novidade que é esta linguagem nos meios académicos, é necessária uma introdução e análise da linguagem e da sua utilização, de modo a servir de base para futuros desenvolvimentos, servindo esta

dissertação como o ponto de partida para o desenvolvimento de processos de negócio/WS usando BPEL.

É apresentado assim uma descrição do contexto da linguagem nas tecnologias de Informação, uma descrição do seu funcionamento e interacção com *Web Services*, assim como os elementos que a constituem, designados por actividades, assim como da sua utilidade dentro de um processo de negócio.

Os três processos mostrados visam sobretudo mostrar a maior parte das actividades presentes na linguagem, explicando-as e mostrando o seu funcionamento. Dai, e depois de se apreender os mecanismos e o funcionamento destes num processo em BPEL, torna-se mais fácil construir novos processos.

Ainda existem muitas lacunas no suporte ao BPEL, especialmente na disponibilização de tutoriais para as aplicações existentes, sendo que mesmo a instalação e integração de alguns dos motores, como no caso do ODE, ainda estão muito sujeitos a incompatibilidades de versões entre o motor BPEL, o *servlet* utilizado (Tomcat ou outros dos aconselhados na sua página oficial), e as versões dos motores Java, sendo que ainda é necessário um certo trabalho de pesquisa em fóruns e FAQ's de apoio de modo a descobrir e resolver essas incompatibilidades. Apesar disso, já se nota uma melhoria na informação disponibilizada online, quer pelas companhias quer por utilizadores individuais

A evolução deste trabalho passa pela análise dos casos apresentados, pela pesquisa de WS em sites da especialidade de modo a encontrar serviços que possam servir as necessidades dos programadores nos contextos que estes estão a trabalhar. Para além da pesquisa, é necessário um certo estudo dos serviços, portos de entrada e saída, tipos de dados que recebem, etc., de modo a que estes possam ser usados numa coreografia de um processo de negócio, ele mesmo um WS pronto a ser consumido no final da sua elaboração.

Outra das vertentes que ainda não tem sido muito focada nesta área refere-se à segurança da utilização e acesso aos WS. Os serviços usados e orquestrados não possuem, na sua generalidade, mecanismos de segurança para lidar com ataques de utilizadores/aplicações maliciosas, seja de DOS (*Denial of Service*) ou de qualquer outro tipo de ataque. Assim, uma área de desenvolvimento possível e necessária quando falamos em WS e na sua

orquestração é a criação de mecanismos de defesa, prevenção e recuperação de ataques, uma vez que, com a proliferação do uso de WS, também as possibilidades de ataques aumentam. Uma vez que esta tecnologia é cada vez mais usada no mundo empresarial, é vital que as empresas que consomem e disponibilizam WS o possam fazer de um modo seguro, protegendo as informações confidenciais e os seus produtos.

Num mundo em que a internet é cada vez mais uma ferramenta de trabalho essencial e incontornável, com os *Web Services* a representarem um dos maiores movimentos de evoluçao, o BPEL é a ferramenta ideal para os dominar, e os seus utilizadores os mais bem preparados para os desafios do presente e do futuro.

Referências

1. Wainewright, P. (2002) *Web Services Infrastructure - The global utility for real-time business*.

2. Diane Jordan, J.E., et al. (2007) *Web Services Business Process Execution Language Version 2.0 - Committee Specification*.

3. Foundation, A.S. *Apache ODE*. [cited 29/09/2007]; Available from: http://ode.apache.org/.

4. *Netbeans*. [cited 17/01/2008]; Available from: http://www.netbeans.org/index.html.

5. Endpoints, A. *ActiveVOS*. [cited 01/02/2008]; Available from: http://www.activevos.com/community-open-source.php.

6. Curtis, D. (1997) *Java, RMI and CORBA*.

7. Harrison, R.C. (1998) *OPC DCOM White Paper*.

8. Group, T.O. *DCE Portal*. 2007 [cited 27/02/2008]; Available from: http://www.opengroup.org/dce/.

9. Microsoft. *MSDN Architecture Center*. 2008 [cited 07/07/2008]; Available from: http://msdn.microsoft.com/en-us/library/bb833022.aspx.

10. Miguel, A. *WS-BPEL 2.0 Tutorial*. 2005 [cited 15/01/2008]; Available from: http://www.eclipse.org/tptp/platform/documents/design/choreography_html/tutorials/wsbpel_tut.html.

11. Microsoft. *DCOM Technical Overview* 2008 [cited 19/03/2008]; Available from: http://technet.microsoft.com/en-us/library/cc722925.aspx.

12. D. Box, D.E., G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, D. Winer (2000) *Simple Object Access Protocol (SOAP) 1.1*.

13. *REST wiki frontpage*. 2008 [cited 28/08/2008]; Available from: http://rest.blueoxen.net/cgi-bin/wiki.pl?FrontPage.

14. Srinivasan, R. (2005) *RPC: Remote Procedure Call Protocol Specification Version 2*.

15. Matsumura, M. (2005) *ESB Roundup Part One: Defining the ESB*. InfoQ

16. Josuttis, N.M., *SOA in practice - Art of Distributed System Design*. 2007: O'Reilly & Assoc.

17. Yan, D.Y., *Service Computing*. 2008.

18. Group, W.C.W. *Web Services Glossary*. 2004 [cited 04/04/2008]; Available from: http://www.w3.org/TR/ws-gloss/.

19. T. Bray, J.P., C. M. Sperberg-McQueen, E. Maler, F. Yergeau (2004) *Extensible Markup Language (XML) 1.0 (Third Edition)*.

20. E. Christensen, F.C., G. Meredith, S. Weerawarana (2001) *Web Services Definition Language (WSDL) 1.1*.

21. L. Clement, A.H., C. V. Riegen, T. Rogers (2004) *UDDI Version 3.0.2*.

22. Inside, S. (2006) *SOA - Arquitectura Orientada a Serviços*.

23. Standards, F.I.P. (1996) *ELECTRONIC DATA INTERCHANGE (EDI)*.

24. Sun Microsystems, I. (1988) *RFC1050 - RPC: Remote Procedure Call Protocol specification*.

25. Sun Microsystems, I. *JavaTM Remote Method Invocation*. 2004 [cited 28/08/2008]; Available from: http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html.

26. UC Irvine, J.G., Compaq/W3C, J. Mogul, Compaq, H. Frystyk, W3C/MIT, L. Masinter, Xerox, P. Leach, Microsoft, T. Berners-Lee, W3C/MIT. *Hypertext Transfer Protocol -- HTTP/1.1*. 1999 [cited 13/03/2008]; Available from: http://www.w3.org/Protocols/rfc2616/rfc2616.html.

27. California, I.S.I.-U.o.S. *RFC 793 - Transmission Control Protocol*. 1981 [cited 09/10/2008]; Available from: http://www.faqs.org/rfcs/rfc793.html.

28. Levitt, J. (2001) *From EDI To XML And UDDI: A Brief History Of Web Services*. InformationWeek

29. Martin Gudgin, M.H., Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, Yves Lafon. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. 2007 [cited 13/11/2007]; Available from: http://www.w3.org/TR/soap12-part1/.

30. Loney, M. (2003) *The state of Web services*. ZDNet.co.uk

31. Team, I.S.A. *Web Services architecture overview*. 2000 [cited 24/10/2007]; Available from: http://www.ibm.com/developerworks/library/w-ovr/.

32. Mertz, D. (2001) *Understanding ebXML - Untangling the business Web of the future*.

33. *RosettaNet*. [cited 02/08/2008]; Available from: http://www.rosettanet.org/cms/sites/RosettaNet/.

34. OASIS. *OASIS Universal Business Language (UBL)*. [cited 24/08/2008]; Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ubl.

35. Rubio, D. (2005) *BPEL: Web Services orchestration, hands-on with ActiveBPEL*.

36. Microsoft (2008) *XLANG/s Language*. MSDN

37. IBM. *WSDL*. [cited 23/08/2008]; Available from: http://www.ebpml.org/wsfl.htm.

38. Oasis. *OASIS - Advancing open standards for the information society*. [cited 16/09/2008]; Available from: http://www.oasis-open.org/home/index.php.

39. Matlis, J. (2005) *Explainer: BPEL*. Computerworld

40. Matjaz Juric, P.S., Benny Mathew *Business Process Execution Language for Web Services*. 2 ed. 2006: Packt Publishing.

41. Brown, P. (2007) *An Introduction to Apache ODE*. InfoQueue

42. Dmitry Markovski, M.E. *Developer Guide to BPEL Designer: The BPEL Runtime*. 2006 [cited 15/12/2007]; Available from: http://www.netbeans.org/kb/55/bpel_gsg.html.

43. Maurer, P. (2006) *Product Review — ActiveBPEL 2.0 from Active Endpoints Excels at BPEL*. SOA World magazine

44. Fionn Murtagh, P.C. (2006) *Orquestration Meeting - Web Services Orchestration*.

45. Sara Graves, R.R., Ken Keiser, Manil Maskey (2007) *Solution Service Composition for Analysis of Online Science Data*. Volume,

46. ActiveEndpoints. *BPEL Samples*. [cited; Available from: file:///C:/Program%20Files/ActiveBPEL%20Designer/Samples/BPEL_Samples/doc/index.html#.

Acrónimos

API – Application Programming Interface

BPEL – Business Process Enterprise Language

CORBA – Common Object Request Broker Architecture

DAO – Data Access Objects

DCOM – Distributed Component Object Model

EBXML – Electronic Business XML

EDI – Electronic Data Interchange

EJB – Enterprise Java Beans

ESB – Enterprise Service Bus

GUI – Graphic User Interfaces

HTTP – Hypertext Transfer Protocol

IDE – Integrated Development Environment

IT – Information Technology

JACOB – Java Concurrent Objects

JBI – Java Business Integration

JDBC – Java DataBase Connectivity

JRMI – Java Remote Method Invocation

OASIS – Organization for the Advancement of Structured Information Standards

OpenJPA – Open Java Persistence Objects

REST – Representational State Transfer

RPC – Remote Procedure Call

SOA – Service-Oriented Architecture

SOAP – Simple Object Access Protocol

UBL – Universal Business Language

UDDI – Universal Description, Discovery and Integration

URPC – Unix Remote Procedure Call

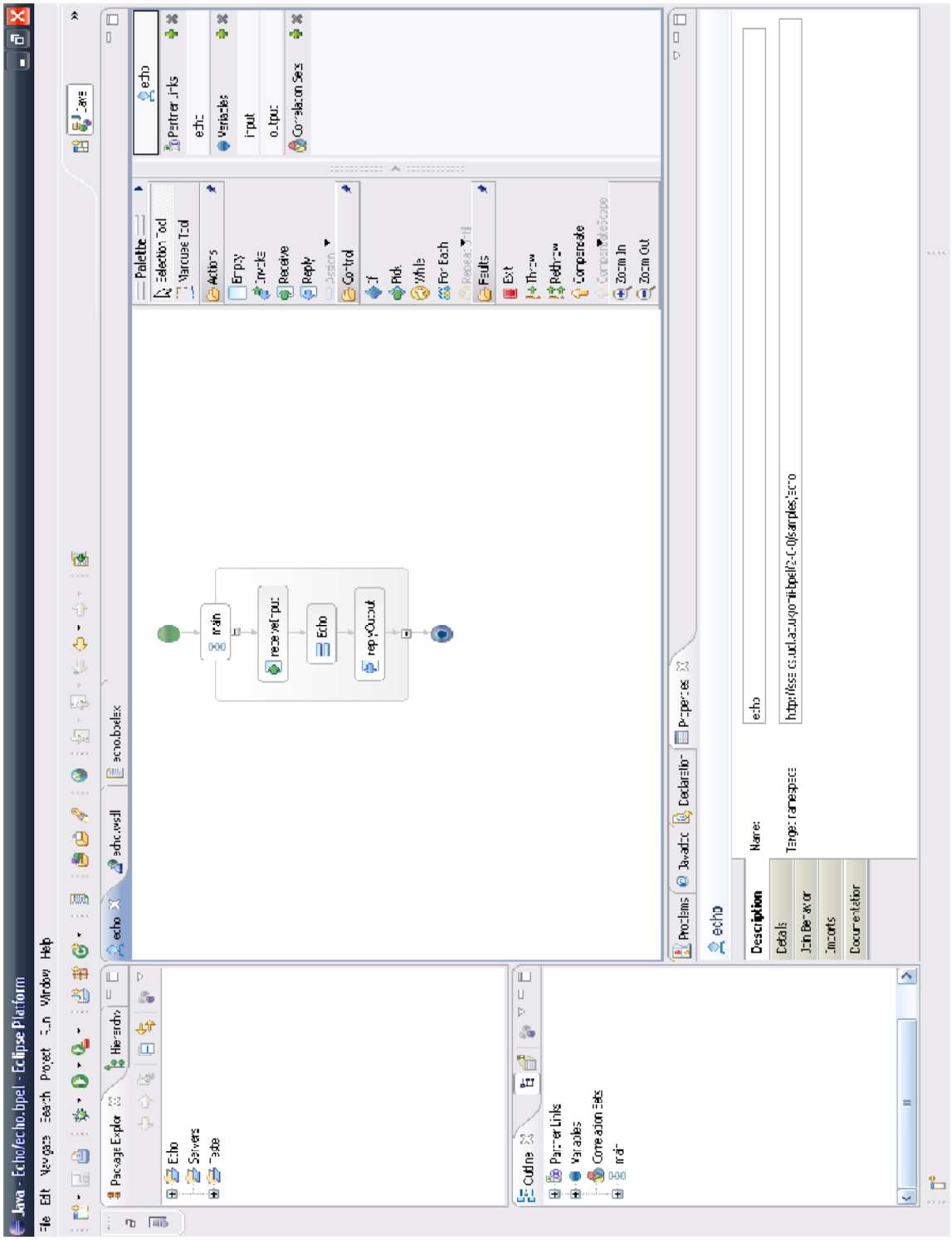
WS – Web Services

WSDL – Web Services Description Language

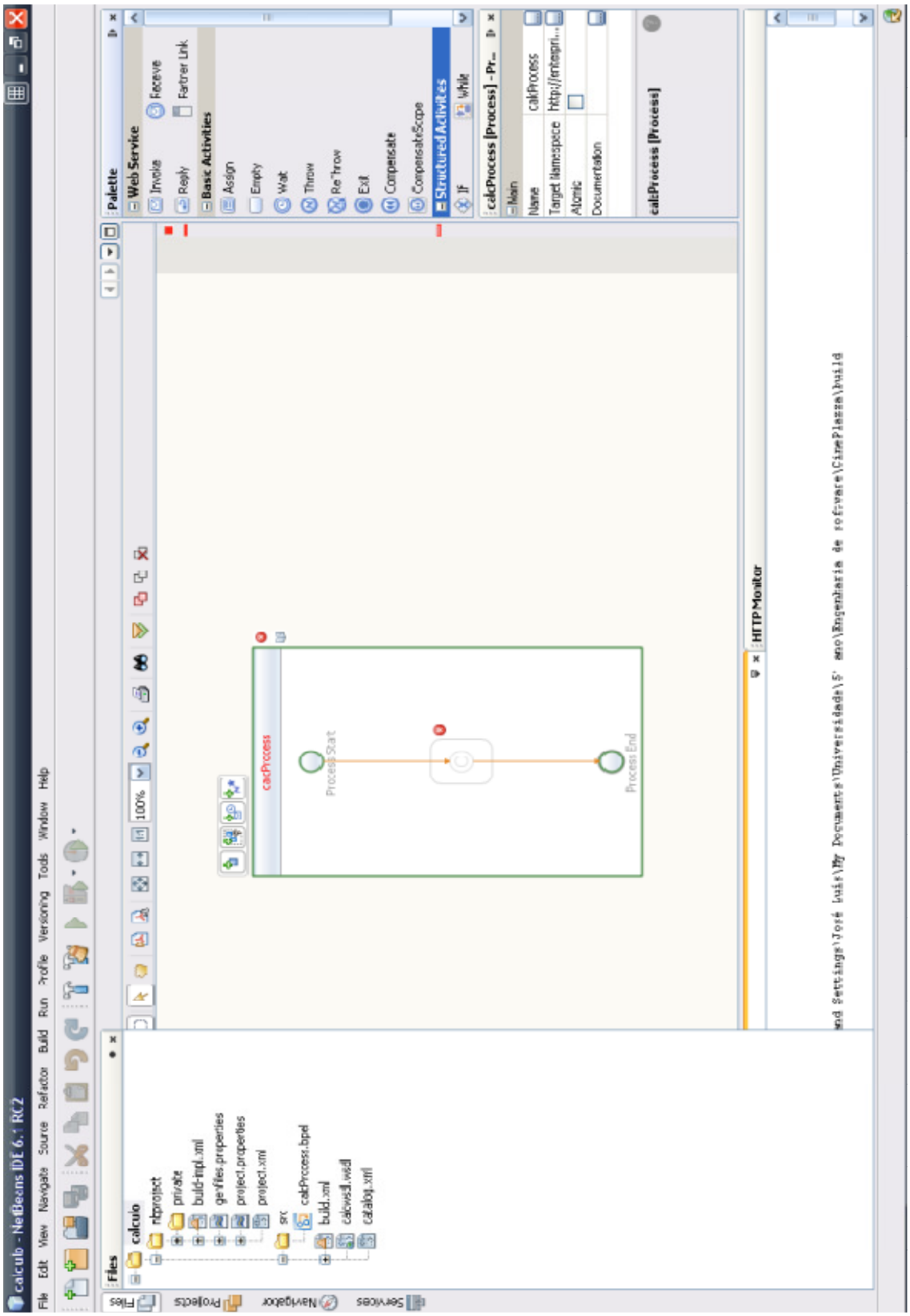
XML – Extensible Markup Language

Apêndices

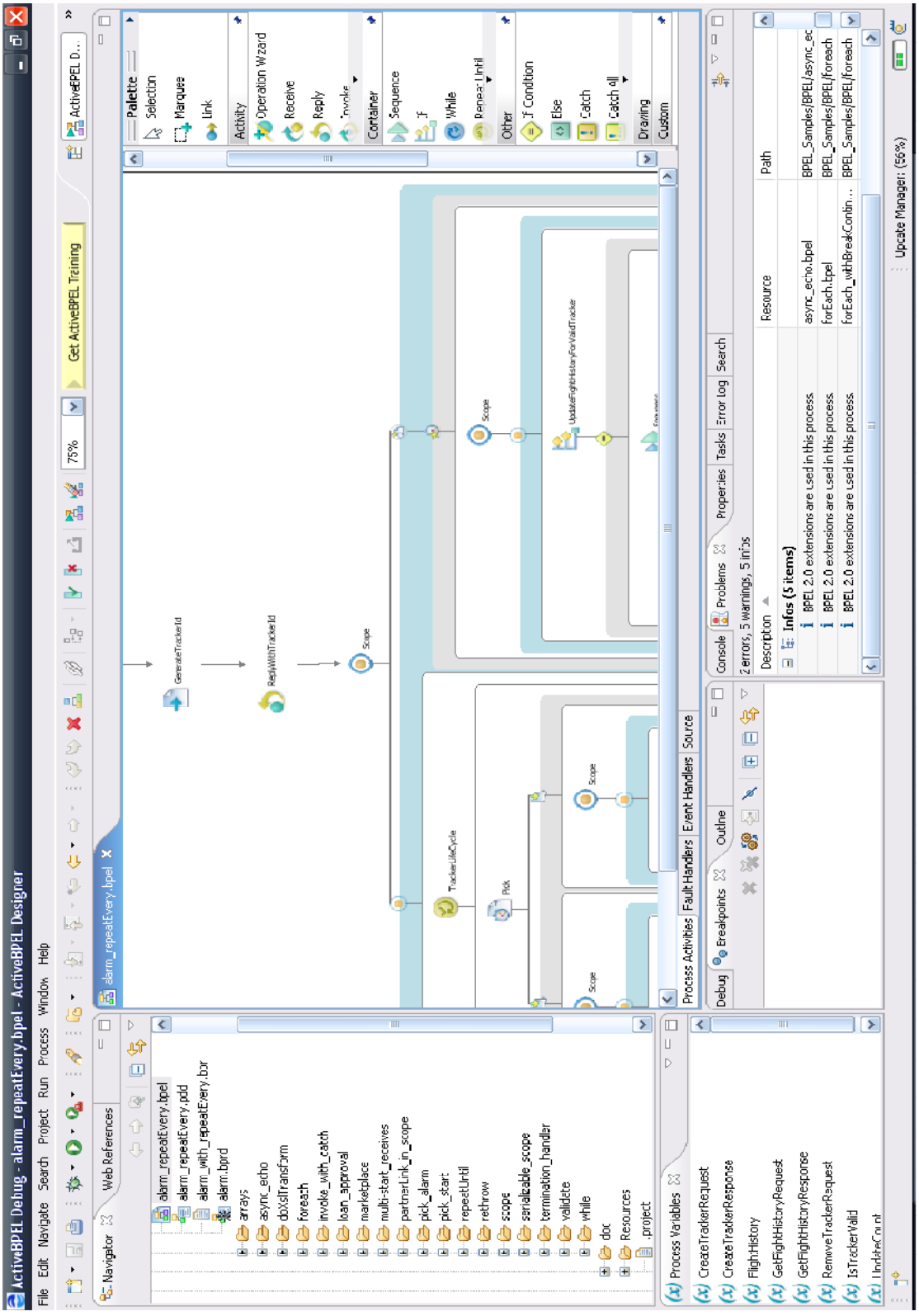
Apêndice 1: Projecto BPEL em Eclipse.



Apêndice 2: Projecto BPEL em Netbeans.



Apêndice 3: Projecto BPEL em activeBPEL. Possui uma aparência semelhante ao *workbench* do Eclipse, uma vez que é baseado neste.



Apêndice 4 : Na consola de administração do motor BPEL podem-se encontrar os processos publicados (*deployed*), assim como os que se encontam em execução, juntamente com várias opções ligadas ao motor e aos estados dos processos e das execuções